Département d'Informatique
Université de PARIS 8 - Vincennes
Route de la Tourelle
75571 Paris CEDEX 12
Tél : 374 12 50 poste 299



V L I S P 8 . 2

M A N U E L D E R E F E R E N C E

Jérôme CHAILLOUX

MARS 1979

Révision 1 (Mars 1980)

TABLE DES MATIERES

AVANT-PROPOS

Le système VLISP 8 . 2, implanté sur micro-processeur de la famille Intel 8080 / Zilog 80, est une version du système VLISP conçu et développé à l'Université de PARIS 8 - VINCENNES.

Le système VLISP est actuellement disponible sur une grande variété d'ordinateurs tels :

- le T1600 de la Télémécanique électrique (sous le système BOS/D)
- le PDP 10 (sous les systèmes TOPS10, WAITS, IRCAM et TENEX)
- le SOLAR 16 (sous le système TSF)
- le PDP11 (sous les systèmes RT11 et RSX11)
- le PDP20 (sous le système TOPS20)

et il est cours de développement sur d'autres ordinateurs parmi lesquels :

- le Motorola 6800
- le CII-HB série 60 level 68 (sous le système MULTICS)

Le langage et le système sont destinés à l'enseignement et à la recherche en programmation expérimentale, en intelligence artificielle, en informatique musicale, en synthèse d'images colorées et en expérimentation d'apprentissage autonome.

Ce manuel décrit ***toutes*** les versions du système VLISP 8 fonctionnant sur micro-processeur de la famille Intel 8080 / Zilog 80. Ce système est disponible dans des configurations très différentes, incluant des systèmes de développements très sophistiqués (tels les systèmes MDS INTELLEC80 ou TRSDOS) ou des systèmes personnalisés les plus variés.

Ce manuel décrit en principe tous les aspects courants des différents systèmes VLISP 8, mais n'engage l'auteur que sur les principes de base. Le manuel sera sujet à des remaniements, au fur et à mesure de la création de nouveaux systèmes VLISP 8 et de l'évolution normale et souhaitée des systèmes actuels et de leurs utilisateurs.

Ce manuel n'est pas réellement destiné aux débutants, mais aux utilisateurs ayant déjà une certaine expérience de LISP. Le lecteur ne doit donc pas espérer trouver une introduction progressive et ordonnée aux constructions du langage. Toutefois la très grande quantité d'exemples de tout niveau doit permettre à ce manuel d'être tout à la fois un manuel de référence et un manuel d'utilisation.



Si vous détectez une situation très anomalique, une erreur manifeste d'un des systèmes, une omission dans le manuel ou si vous désirez parler de choses et d'autres, contactez-moi à l'adresse ci-dessous :

Jérôme CHAILLOUX
Département d'Informatique
Université Paris-8 Vincennes
Route de la Tourelle
75571 Paris Cédex 12. FRANCE
tél : 374-12-50 Poste 299

Je crois à propos de remercier P. GREUSSAY, H. WERTZ, D. GOOSSENS, K. ZELASKA, J.F. PERROT, B. ROBINET, H. HUITRIC, M. NAHAS, G. PAUL, A. CATTENAT, L. AUDOIRE, J.P. MOULIN, C. COLERE, Y. DEVILLER, G. ENGLERT, Y. LECERF, J.M. HULLOT, G. HUET, G. BERRY, D. RONCIN, G. COSLADO, J.C. RAOULT, G. HERBUVEAUX, M. SAINTOURENS, G. NOWAK
ainsi que tous les autres utilisateurs qui ont influencé, par leurs excellentes suggestions, la réalisation et l'implémentation des différents systèmes VLISP 8.
En particulier, que tous les membres du Club de Micro-Informatique, ARRAKIS, (de l'Université de Paris 8) soient remerciés pour leur collaboration tant matérielle que logicielle.

La réalisation de la version MDS a été possible grâce à l'intérêt et à la bienveillance de Jean Paul MAZEAU.

La réalisation de la version SORCERER a été rendue possible grâce à la compétence de Gérard BERRY.

La réalisation des systèmes TRS a été possible grâce au soutien de Gerald BENETT et de Jean KOTT, à l'IRCAM.

La composition de ce manuel a été réalisée à l'IRCAM avec le programme RF de J.L. RICHER, et l'impression du fichier édité a été réalisée sur l'imprimante électrostatique VERSATEC grâce à la compétence de R. BARA et P. GREUSSAY.

La révision 1 de ce manuel reflète les nombreuses améliorations apportées à l'interprète VLISP durant l'année 1979. Une description complète de ce nouvel interprète est donnée dans :

Le modèle VLISP :
Description, Implémentation
et Evaluation.

Jérôme CHAILLOUX,
Thèse de 3ème cycle,
Université Pierre et Marie Curie,
L. I. T. P., 2 Place Jussieu
75221 Paris Cédex 05 France
Avril 1980

PLAN DE LECTURE

Ce manuel se veut être un ***manuel de référence*** et risque de paraître obscur à ceux qui désirent une introduction progressive aux concepts et aux structures du langage. Toutefois l'abondance d'exemples doit permettre à ce manuel d'être également un ***manuel d'utilisation***. Ce manuel s'adresse donc aux lecteurs ayants déjà une petite notion du langage LISP.

Le chapitre 1 contient la description des différents systèmes ainsi que leur mise en oeuvre. Sa lecture est indispensable pour un premier contact avec le système et pour l'adapter à un système différent.

Le chapitre 2 traite de la représentation des données et du fonctionnement interne de l'interprète. Ce chapitre, notoirement opaque, peut être sauté en première lecture par les utilisateurs ayants déjà une idée de l'interprétation de LISP en général. Ne s'y référer que pour les détails d'implémentation et de l'interprétation de VLISP.

Le chapitre 3 contient la description des fonctions standards des différents systèmes. ATTENTION : ***toutes*** ces fonctions ne se trouvent pas forcément sur ***tous*** les systèmes. En effet certaines fonctions ne sont pas incluses dans les ***petits*** systèmes VLISP 8.

Le chapitre 4 traite des entrées/sorties simples et de la manipulation des fichiers. Là encore du fait des périphériques spécialisés toutes les fonctions de ce chapitre ne se trouvent pas sur tous les systèmes.

Le chapitre 5 traite des fonctions systèmes. Sa lecture est reservée à ceux désirant réaliser des systèmes spéciaux.

Le chapitre 6 décrit l'éditeur vidéo interne EDITV du système TRS VLISP 8. A lire très vite si votre sytème possède un tel éditeur.

Le chapitre 7 contient la description et le texte du paragrapheur VLISP plus connu sous le nom de PRETTY-PRINT, que possèdent les systèmes à disquettes.

Le chapitre 8 donne le texte de programmes VLISP utilitaires et de démonstration.

Enfin une annexe de Harald WERTZ introduit le système LOGO/LISP. Toutes les primitives du langage y sont décrites ainsi que de très nombreux exemples.

I - UTILISATION

1.1 LES DIFFERENTES VERSIONS DU SYSTEME VLISP 8.

Le système VLISP 8.2 utilise indifféremment une unité centrale de type Intel 8080 ou Zilog 80. Les systèmes VLISP utilisant le Zilog 80 sont à la fois plus rapides et moins encombrants (du fait de son jeu d'instructions amélioré), mais dans une proportion négligeable ce qui permet d'assimiler ces deux types de C.P.U.

Il existe deux grands types de systèmes VLISP 8.2 :
- ceux qui utilisent des systèmes de développements existants
- et ceux qui utilisent des micro-ordinateurs de fabrication personnalisée.

Dans tous les cas l'interprète lui-même occupe une place fixe de 8k à 12k octets (en fonction des systèmes) et demande un minimum de 8k octets de mémoires de travail. Cette mémoire de travail doit être de l'ordre de 16k pour tout travail conséquent.

1.1.1 VLISP et les systèmes de développements

Le système VLISP 8.2 est disponible sur les systèmes de développement suivants :

MDS Intellec 80 sous système disque ISIS 1 ou ISIS II, avec une mémoire centrale de 32k ou de 64k. Ce système est chargé à partir du disque souple et laisse 10k de mémoire de travail pour l'interprète dans le version 32k et 42k de mémoire de travail dans le version 64k. Dans ce manuel nous appelerons ces deux versions MDS(32k) ou MDS(64k). La configuration peut comporter beaucoup de périphériques qui seront tous utilisables en VLISP par l'intermédiaire de ISIS.

TRS80 sous système Level II, , avec un minimum de 16k RAM et optionnellement, avec adjonction d'une carte contenant 8k REPROM. La réalisation de la carte supplémentaire adaptable sur le bus standard du TRS80 a été réalisée par Christian COLERE. Dans le système sans REPROM, l'interprète doit être chargé à partir d'une cassette et la mémoire de travail de l'interprète ne dépasse pas 8k; en revanche dans la version avec REPROM la taille de la mémoire de travail est de 16k. Nous nommerons ces versions TRS80(K7) et TRS80(PROM) tout au long de ce manuel.
Il existe une autre version de VLISP fonctionnant sur TRS80 avec une grosse configuration qui comporte au minimum un floppy disque et 48k de mémoire vive. Cette version utilise un système de gestion de disquette qui peut être : TRSDOS, NEWDOS+ ou bien CP/M. Nous nommerons tout au long de ce manuel TRS80(TRSDOS) la version fonctionnant sous système TRSDOS ou NEWDOS+ et TRS80(CP/M) la version

fonctionnant sous CP/M.
Avec ces quatre familles du système VLISP 8 fonctionnant sur TRS80, il est possible d'utiliser le lecteur de cassette, les possibilitées pseudo-graphiques de l'écran et un éditeur temps réel sur écran inclu dans le système VLISP 8 lui-même.

SORCERER dans 3 versions utilisant respectivement :
- là cassette, système SORCERER (K7)
- le ROM PAC, système SORCERER (ROMPAC)
- la disquette sous système CP/M, système SORCERER (CP/M)

Tous ces systèmes utilisent 32k ou 48k RAM et permettre d'acceder à l'interface série. Un éditeur temps réel sur écran est en cours d'écriture.

1.1.2 VLISP et les systèmes personnalisés

A l'heure actuelle VLISP est implanté sur deux systèmes personnalisés conçus autour des modules :

SDK 80 de INTEL : à base de 8080 qui utilise 16k RAM et 8k REPROM, Ce système a été réalisé par J.P. MOULIN. Nous appelerons cette version SDK80 tout au long de ce manuel.

SDB 80E de MOSTEK : à base de Z80 qui utilise également 16k RAM et 8k PROM. Ce système a été réalisé par Louis AUDOIRE qui a également ajouté de nombreux dispositifs annexes en particulier une unité arithmétique de type AMD9511. Nous appelerons cette version MZ80.

On peut bien évidemment étendre la mémoire de ces différents systèmes, leur ajouter à tous une Unité Arithmétique de type Amd9511 et de nombreux périphériques.



1.2 POUR DEBUTER AVEC VLISP 8.

Pour jouer VLISP 8, il suffit, sous moniteur, d'émettre sur le terminal la commande :

```
-VLISP      dans les systèmes MDS (32k) et MDS (64k)

.E 8020     dans le système MZ80

.G8020      dans le système SDK80

>SYSTEM puis
?/32800      dans le système TRS80 (PROM)

>SYSTEM puis
?/16400      dans le système TRS80 (K7)

DOS READY
VLISP        dans le système TRS80 (TRSDOS)

LO VLISP puis
GO 0420      dans le système SORCERER (K7)
```

dans tous les cas, le système répond :

```
** VLISP 8.2  jj/mm/aa  xxxxxxx
** (C) 1979 Université de Paris 8 - Vincennes
```

dans lequel jj/mm/aa est la date de la dernière modification de l'interprète et xxxxxx est le nom du système employé. Le système rentre alors dans la boucle principale de l'interprète qui va, indéfiniment, lire une expression sur le terminal, l'évaluer puis imprimer la valeur de cette évaluation. Le système indique qu'il attend la lecture d'une expression en imprimant, sur le terminal, le caractère ? au début de chaque ligne. Enfin la valeur de l'évaluation est imprimée précédée du caractère =.

La fin de ce chapitre contient trois exemples de session simple avec différents systèmes VLISP 8.2.

1.3 IMPLANTATION ET INTERFACE D'ENTREE/SORTIE.

L'implantation mémoire est fonction du système utilisé. Toutefois la réalisation d'interfaces d'entrée/sorties spéciaux est extrêmenet aisé. En effet, l'interprète débute par une table de branchements sur les différents points de lancement de l'interprète ainsi que sur les modules d'entrée/sortie normaux et auxilliaires.

Voici l'organisation de la table des branchements de l'interprète.

Lancement de l'interprète à froid.	START:	JMP	xxx
Lancement de l'interprète à chaud.	WARM:	JMP	xxx
Entrée console. (retourne dans le registre A le caractère suivant lu sur la console).	CI:	JMP	xxx
Test console. (retourne dans le registre A : 0, si aucun caractère n'est prêt à être lu, sinon le caractère lu).	CS:	JMP	xxx
Sortie console. (sort sur la console le caractère contenu dans le registre C).	CO:	JMP	xxx
Entrée auxilliaire. (équivalent à CI: pour le périphérique auxilliaire).	TR:	JMP	xxx
Sortie auxilliaire. (équivalent à CO: pour le périphérique auxilliaire).	TP:	JMP	xxx

Connaissant le début de cette table, il est aisé de changer l'adresse de branchement de l'un de ces modules.

Cette table se trouve en :

à l'adresse :	pour le système :
3420	MDS (32k)
3820	MDS (64k)
8020	MZ80
4C20	TRS80 (K7)
8020	TRS80 (PROM) inaltérable !
8020	TRS80 (TRSDOS + 32k)
9020	TRS80 (TRSDOS + 48k)
8020	SDK80
0420	SORCERER (K7)
0420	SORCERER (CP/M + 48k)

Exemple de session simple de VLISP avec un système MDS (64k)

```
-VLISP                             ; Sous ISIS II appel du système VLISP

** VLISP 8.2   14/9/79  MDS (64k)            ; répond le système qui
** (C) 1979 Universite de Paris 8 - Vincennes

?                                  ; attend des données sur le terminal
?  ()                              ; NIL est équivalent à ()
=  NIL                             ; dit l'interprète.
?
?  567                             ; La valeur d'un nombre est ce nombre
=  567                             ; lui-même.
?
?
?  (CAR '(A B C))                  ; Appel plus complexe n'est-ce-pas ...
=  A
?
?  (cdr '(a b c))                  ; Les caractères minuscules sont automatiquement
=  (B C)                           ; convertis en majuscules
?
?  (DE FOO (A L)                   ;
?     (COND
?        ((NULL L) 0)
?        ..                        ; .. produit irrémédiablement une erreur
                                   ; de syntaxe et re-entre dans la boucle
 ** ERREUR DE SYNTAXE.             ; de lecture normale.
 ** BREAK
?
?  (DE FOO (A L)                   ; Définition de la fonction qui
?     (COND                        ; retourne le nombre d'occurrences
?        ((ATOM L) 0)              ; de l'atome A dans la liste L.
?        ((EQ (CAR L) A) (ADD1 (FOO A (CDR L))))
?        ((FOO A (CDR L))))))))
=  FOO                             ; La fonction est maintenant définie.
?
?  (FOO 'DO '(DO DO DO RE MI RE DO MI RE RE DO))  ; Et elle fonctionne.
=  5
?
?                                  ;  ... etc ...
?  (END)                           ; pour arrêter l'interprète et repasser
                                   ; sous le contrôle de ISIS.

ISIS V3.4                          ; répond ISIS
-
```

Exemple de session simple de VLISP avec un système TRS80 (PROM)

```
>                                       ; sous le LEVEL II
>SYSTEM                                 ; appel de l'interprète

*? /32800

** VLISP 8.2  14/9/79  TRS80 (PROM)                     ; répond VLISP 8
** (C) 1979 UNIVERSITE DE PARIS 8 - VINCENNES

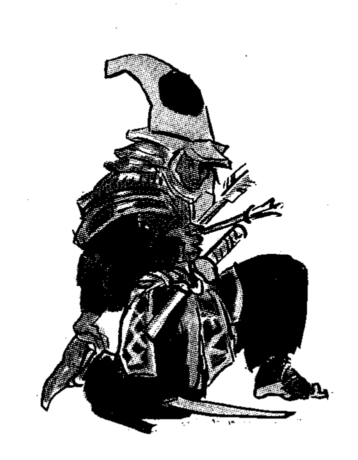
?
?  (DE POL (E)                          ; Définition d'une fonction de
?     (IF (ATOM E) E                    ; polonisation préfixe d'une
?        (LIST (CADR E) (POL (CAR E)) (POL (CADDR E)))))
=  POL                                  ; expression 2-aire
?
?  (POL '(A * B))                       ; Essai de la fonction.
=  (* A B)
?
?  (POL '((A + B) * (A - B)))
=  (* (+ A B) (- A B))
?
?  (POL '((B ↑ 2) - (4 * (A * C)))))
=  (- (↑ B 2) (* 4 (* A C)))
?
?
? ; exemple de fonctions graphiques utilisant les
? ; fonctions sur écran : CLEAR et DISPLAY :
?
? (DE ANIM (N POS L1 L2 L3)
?    (IF (= N 0)
?       ()
?       (DISPLAY  POS L1)
?       (DISPLAY (+ POS 64) L2)
?       (DISPLAY (+ POS 128) L3)
?       (ANIM (1- N) (1+ POS) L1 L2 L3)))
= ANIM
?
? (DE ELEF (N POS)
?    ; dessine un éléphant !
?    (ANIM N POS
?       '(#20 #20 #B0 #BC #BC #B4 #B8 #BC #90)
?       '(#20 #96 #BF #BF #BF #BF #9F #86 #BD)
?       '(#20 #81 #BF #95 #20 #BF #95 #20 #8A)))
= ELEF
?
? (DE VOIR ()
?    (CLEAR #20)
?    (ELEF 30 64)
?    (ELEF 60 320)
?    (ELEF 40 512)
?    (ELEF 20 640)
?    (IF (= (TYI) #20) (VOIR)))
= VOIR
```

```
?
? (VOIR)

  ..... Pour voir, il faut appuyer sur
  ..... la barre d'espacement.
  ..... Pour interrompre le programme il
  ..... faut appuyer sur la  touche BREAK

  ** BREAK
?
? (END)                  ; pour revenir au LEVEL II

MEMORY SIZE?             ; dit-il
>READY
```



Exemple de session simple de VLISP avec un système MZ80

```
.                                    ; sous DDT 80
.E 8020                              ; appel de l'interprète

** VLISP 8.2   14/9/79  MZ80                ; répond VLISP 8
** (C) 1979 Universite de Paris 8 - Vincennes

?
? (COULT 0)               ; effacement de tout l'écran couleur
= 0

   ; Définition de quelques fonctions utilisant l'écran couleur (1)

?
? (DE SEG (x0 y0 x1 y1 x)
?    (ADD y1 (DIV (MUL (SUB y1 y0) (SUB x x1))
?                 (SUB x1 x0))))
= SEG
?
?
? (DE VOIR (m1 n1 m2 n2 m3 n3 ;; I J)
?    (ADRIX 0)
?    (SETQ J 0)
?    (WHILE (LE J 56)
?       (SETQ I 0)
?       (WHILE (LE I 63)
?          (COULDOSE
?             (SEG 0 M1 63 N1 I)
?             (SEG 0 M2 63 N2 I)
?             (SEG 0 M3 56 N3 J))
?          (SETQ I (ADD1 I)))
?       (SETQ J (ADD1 J))))
= VOIR
?
? (VOIR 0 15 15 0 0 15)
= NIL
?
? (VOIR 0 15 0 15 0 15)
= NIL
?
? (END)                    ; pour repasser sous le contrôle de DDT80

.                          ; dit-il
```

(1) ce programme réalise une série continue très simple, voir à ce sujet ***Les séries continues colorées*** , par Monique NAHAS et Hervé HUITRIC, Février 1979, Département d'Informatique, Université de Paris 8 - Vincennes, Route la Tourelle, 75571 Paris Cédex 12 FRANCE.

II - L'INTERPRETE

2.1 LES OBJETS VLISP 8

L'interprète VLISP 8.2 permet de manipuler des objets nommés *S-expressions* (pour Symbolic expressions).

Ces objets sont classés en 4 types distincts :
- les atomes littéraux
- les atomes numériques
- les listes
- les tableaux d'objets binaires (de code).

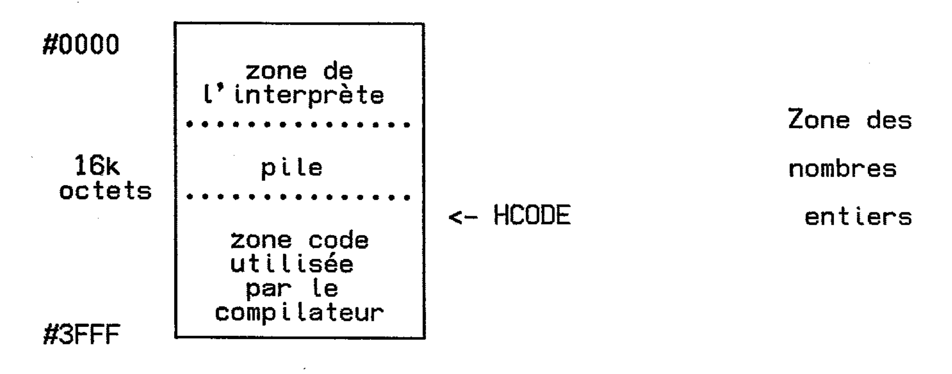
Toute S-expression est représentée en machine au moyen d'un pointeur (ou d'une adresse).
L'accès aux valeurs des différents objets sera donc toujours effectué au moyen d'une indirection.

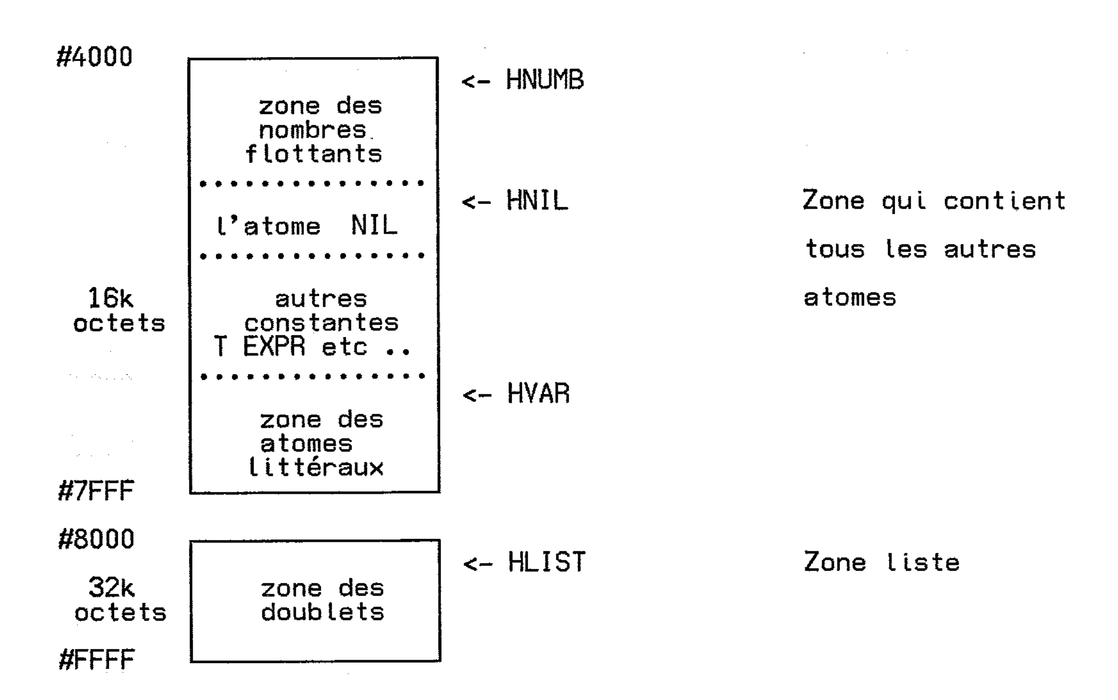
VLISP sera donc spécialisé dans la manipulation de pointeurs.

2.1.1 Organisation de la mémoire

L'organisation de la mémoire est différente pour chacun des systèmes VLISP 8. Toutefois ces organisations ont en commun le découpage de la mémoire en zones fixes ce qui permet de distinguer aisément le type des différents objets et ce qui facilite le travail de l'allocateur/récupérateur de mémoire.

Voici une des organisation possible de la mémoire (chaque système ayant une organisation particulière) :





2.1.2 Les atomes littéraux

Ils jouent le rôle d'identificateurs et servent à nommer les variables et les fonctions. Ils sont créés implicitement dès leur lecture dans le flux d'entrée ou explicitement par les fonctions IMPLODE ou GENSYM ; nul besoin donc de les déclarer.

Leur nom externe (Print name ou P-NAME) est une suite de caractères quelconques (contenant au moins un caractère non numérique) de longueur limitée à 62 caractères. On peut insérer dans un P-NAME des délimiteurs s'ils sont précédés du caractère / (slash) ou mieux, si le P-NAME contient des caractères spéciaux, on peut encadrer tout le P-NAME avec le caractère guillemets (voir la section 4.2.2 sur la lecture standard).

Un tel atome littéral est représenté dans l'interprète par un pointeur sur un descriptif stocké dans une zone spéciale de la mémoire.

Ce descriptif est constitué des 8 ***propriétés naturelles*** suivantes :

C-VAL (abréviation de Cell-VALue) sur 2 octets qui contient à tout moment la valeur LISP associée à l'atome littéral considéré comme une variable. L'accès à cette valeur est extrêmement rapide. A la création d'un atome littéral, sa C-val est "indéfinie". Toute tentative de consultation d'un atome littéral qui n'a pas encore reçu de valeur provoque irrémédiablement une erreur.

P-LIST (abréviation de Properties LIST) sur 2 octets qui contient à tout moment la liste des propriétés de l'atome littéral. Ces propriétés sont gérées par l'utilisateur au moyen des 5 fonctions

sur les P-LIST (PLIST, PUT, GET, REMPROP et ADDPROP). Le système n'utilise jamais les P-LIST des atomes littéraux pour ses besoins propres.

F-VAL (abréviation de Function-VALue) sur 2 octets qui contient à tout moment la valeur associée à l'atome littéral considéré comme une fonction. Cette valeur peut-être :
- une adresse dans le cas des SUBR
- une S-expression dans le cas des EXPR
L'accès à la F-VAL est effectué au moyen de la fonction FVAL et des fonctions de définition DE/DF/DM/DMC.

F-TYPE (abréviation de Function TYPE) sur 1 octet qui contient le type (codé) de la fonction stockée dans la F-VAL. L'ensemble F-VAL, F-TYPE permet de lancer les fonctions d'un manière extrêmement rapide. La consultation, en clair, du F-TYPE des atomes littéraux est réalisée au moyen de la fonction FTYPE.

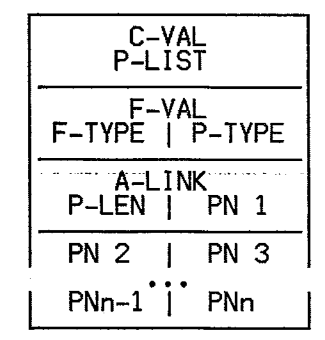
P-TYPE (abréviation de Print TYPE) sur 1 octet qui contient les informations nécessaires à l'édition de la représentation externe de l'atome littéral
- comme variable : le bit 7 est positionné automatiquement par la fonction de lecture à l'entrée d'une pseudo-chaîne
- comme fonction : les 4 bits de poids faibles servent au PRETTY-PRINT pour connaître le format d'édition à utiliser.
L'accès au P-TYPE d'un atome littéral est effectué au moyen de la fonction spéciale PTYPE.

A-LINK (abréviation de Atom-LINK) sur 2 octets qui contient l'adresse de l'atome suivant dans la table des atomes. Ce lien (obligatoire du fait de la taille variable des descriptifs des atomes littéraux) permet entre autre de gérer facilement le hash-coding de la table des atomes. Cet attribut n'est pas accessible à l'utilisateur directement.

P-LEN (abréviation de Print-LENgth) sur 1 octet qui contient le nombre de caractères du P-NAME de l'atome littéral. Le P-LEN est utilisé en conjonction avec le P-TYPE pour contrôler l'édition d'un atome littéral.

P-NAME (abréviation de Print-NAME) sur N octets qui contient les caractères du nom de l'atome littéral.

Ces propriétés naturelles sont rangées en mémoire suivant le schéma :



←——— pointeur sur l'atome littéral

En VLISP 8 le CAR d'un atome littéral est sa C-VAL, le CDR sa P-LIST.

Certains atomes sont déjà connus de l'interprète :
- les constantes littérales (qui contiennent leur propre adresse en C-VAL) dont voici la liste : NIL T LAMBDA 0SUBR 1SUBR 2SUBR 3SUBR NSUBR FSUBR EXPR FEXPR MACRO & QUOTE).
- les fonctions standards

2.1.3 Les nombres

VLISP 8 manipule des nombres entiers (permettant de calculer dans l'intervalle :
- [-2↑13 - 1 , +2↑13] dans les systèmes MDS(32k) et TRS80(K7)
- [-2↑14 - 1 , +2↑14] dans tous les autres systèmes.

et des nombres flottants sur 32 bits dans certains systèmes TRS80(TRSDOS) et TRS80(CP/M).

Le P-NAME d'un nombre est la représentation de sa valeur dans la base de conversion courante (e.g. 10).
La valeur d'un nombre est ce nombre lui-même. Un nombre n'a ni C-VAL ni P-LIST.

2.1.4 Les chaînes de caractères

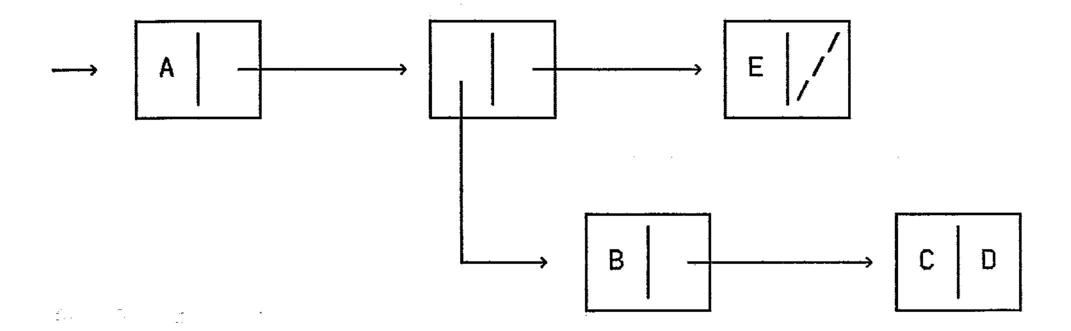
Il n'existe pas à proprement parler de chaînes de caratères, mais des pseudo-chaînes (qui ne sont qu'une autre manière de décrire les atomes littéraux).
Une pseudo-chaîne est une suite de caractères quelconques encadrée du caractère délimiteur de chaîne, le guillemet ".
Ces pseudo-chaînes ont donc les mêmes propriétés que les atomes littéraux, toutefois à leur création elles sont considérées comme des constantes et possèdent donc leur propre valeur en C-VAL. Il n'est donc pas obligatoire de "quoter" ces pseudo-chaînes.

2.1.5 Les listes

Les listes sont représentées d'une manière standard : les différents éléments d'une liste utilisent un doublet de pointeurs dont la partie gauche (le CAR) contient l'élément, et la partie droite (le CDR) contient un pointeur sur l'élément suivant ou NIL pour le dernier élément.



Ex : la liste (A (B C . D) E) est stockée en mémoire
sous la forme :



2.1.6 Le code

Il existe un certain nombre de fonctions spéciales (de manipulation de la mémoire) qui permettent d'utiliser n'importe quelle zone de mémoire. Cette mémoire peut être considérée comme un tableau d'objets binaires.

2.2 FONCTIONNEMENT DE BASE DE L'INTERPRETE

2.2.1 Evaluation des atomes

La valeur d'un atome littéral (considéré comme une variable) est sa C-VAL. L'évaluation d'un atome littéral dont la C-VAL est indéfinie (auquel on a pas encore donné de valeur) provoque lors de son évaluation une erreur dont le libellé est :

***** erreur variable indéfinie : <at>   ou***
***** undefined variable : <at>***

dans lequel le nom de l'atome littéral incriminé ***<at>*** est imprimé.

Il existe de fait 2 types d'utilisation des variables :
- comme variables globales. Ces variables ont une valeur toujours accessible.
- comme variables locales. Danc ce cas les variables reçoivent une valeur qu'elles ne gardent que le temps de l'exécution d'une fonction (ce sont les paramètres formels des fonctions).

La valeur d'une pseudo-chaîne est cette pseudo-chaîne elle-même, nul besoin donc de la "quoter".

La valeur d'un nombre est ce nombre lui-même.

2.2.2 Evaluation d'une liste

L'évaluateur considère toujours une liste comme un appel de fonction. Cette liste s'appelle une forme. Le CAR de cette forme est la fonction, le CDR de la forme les arguments de la fonction.
La valeur d'une forme est la valeur retournée par l'application de la fonction aux arguments.

2.3 LES FONCTIONS

Une fonction (le CAR d'une forme) peut être n'importe quelle S-expression. Le CDR de la forme est la liste des paramètres actuels de la fonction.

Si la fonction est un atome littéral, la fonction à utiliser est celle qui a été associée à cet atome littéral
- soit à l'initialisation du système (c'est le cas des fonctions prédéfinies qui sont appelées également fonctions standards)
- soit par l'utilisateur au moyen des fonctions de définition statiques ou dynamiques.
Si aucune fonction n'a été associée à cet atome, une erreur apparait dont le libellé est :

***** erreur fonction indéfinie : <at>*** ***ou***
***** undefined function <at>***

dans lequel le nom de l'atome littéral incriminé ***<at>*** est imprimé.

Dans le cas où la fonction est un nombre, cela correspond à l'appel implicite de la fonction standard CNTH. Donc l'évaluation de (<n> <l>) retourne le <n>ième élément de la liste 1er argument <l>.

```
ex :  (3 '(A B C D))     ☞  C
      (8 '(1 2 3 4 5)    ☞  NIL
```

Dans le cas où la fonction est une liste, il peut s'agir :
- d'une déclaration explicite et anonyme d'une fonction. Dans ce cas le premier élément de la liste doit être soit l'atome spécial INTERNAL, soit l'atome spécial LAMBDA.
- d'une nouvelle forme qui doit être évaluée et dont la valeur est la fonction à utiliser. On a donc affaire dans ce cas non pas à des fonctions constantes (anonymes ou non) mais à des fonctions variables. VLISP est l'un des rares langages dans lequel il est possible d'écrire des appels très agréables du genre :

```
((IF (GT n 0) 'MUL 'DIV) val 2)
```

VLISP 8 possède 6 familles de fonctions :
- les SUBR et les FSUBR écrites en langage machine.
- les EXPR, les FEXPR et les MACRO écrites en VLISP.
- les fonctions d'échappement de type ESCAPE.

2.3.1 Les SUBR

Les SUBR sont des fonctions écrites en ***langage machine***, résidentes dans l'interprète dès son lancement (les fonctions standards) ou après compilation (les fonctions compilées). Ces fonctions possèdent des ***arguments évalués***. Il existe des SUBR à 0, 1, 2, 3 ou N arguments. Pour les SUBR à nombre fixe d'arguments (0, 1, 2 ou 3) s'il y a trop d'arguments fournis à l'appel, ils sont ignorés SANS être évalués. L'interprète n'évalue donc que le nombre d'arguments dont il a besoin. D'autre part s'il manque des arguments ils sont supposés être liés à la valeur NIL.
Il n'est pas possible de définir des fonctions de ce type, sauf à les écrire directement en LAP ou en faisant compiler des EXPR.
Ces fonctions sont très nombreuses dans l'interprète standard, entre 100 et 150 en fonction du système utilisé.

```
Ex : (CONS 'A) = (CONS 'A NIL) ☞ (A)
     (CONS (PRIN 'A) (PRIN 'B) (PRIN 'C)) A B ☞ (A . B)
```

On peut décrire anonymement une fonction de ce type au moyen d'une liste de la forme :

(INTERNAL <type> <adresse>)

dans laquelle **<type>** est un des types de SUBR parmi les types suivants : 0SUBR 1SUBR 2SUBR 3SUBR ou NSUBR
et **<adresse>** est l'adresse d'implantation du sous-programme écrit en langage-machine réalisant la fonction.

ex : si l'adresse du sous-programme CAR écrit en langage machine se trouve à l'adresse 18715, les deux appels suivants sont équivalents :

```
(CAR '(A B C))                     ☞ A
((INTERNAL 1SUBR 18715) '(A B C))  ☞ A
```

2.3.2 Les FSUBR

Les FSUBR sont également des fonctions écrites en ***langage machine***, résidentes dans l'interprète dès son lancement (les fonctions standards). Ces fonctions possèdent des arguments en nombre variable ***qui ne sont pas évalués***. Ces fonctions sont principalement utilisées comme fonctions de contrôle de l'interprète ou comme fonctions de manipulation de noms. Elles sont peu nombreuses.
Tous comme pour les SUBR il n'est pas possible de définir des fonctions de ce type, sauf à les écrire directement en LAP ou en faisant compiler des FEXPR.
Toutefois il est possible de décrire de telle fonctions anonymement en utilisant une liste de la forme :

(INTERNAL FSUBR <adresse>)

dans laquelle **<adresse>** est l'adresse du sous-programme écrit en langage machine qui réalise la fonction.

ex : si l'adresse du sous-programme qui réalise la fonction NEWL est 28841, les deux appels suivants sont équivalents :

```
(NEWL L)
((INTERNAL FSUBR 28841) L)
```

2.3.3 Les EXPR

Les EXPR sont des fonctions écrites en VLISP et interprétées par les fonctions standards d'évaluation (EVAL et APPLY). Ces fonctions possèdent un nombre quelconque de paramètres formels, qui sont rangés dans une liste **<lvar>** ainsi qu'un corps de fonction constitué d'un nombre quelconque d'expressions **<s1>** ... **<sN>**.

On décrit anonymement une fonction de ce type en utilisant une liste de la forme :

(INTERNAL EXPR (<lvar> <s1> ... <sN>)) ou bien

(LAMBDA <lvar> <s1> ... <sN>)

dans laquelle **<lvar>** est la liste des paramètres formels et **<s1>** ... **<sN>** le corps de la fonction.
La description d'une EXPR sous forme d'une lambda-expression a été conservée pour des raisons historiques (en effet LISP provient du λ-calcul) et des raisons de compatibilité entre les différents dialectes de LISP.
La définition des fonctions de ce type est décrite dans la section suivante.

L'évaluation d'un appel de fonction de type EXPR s'opére en 3 étapes :
1 - liaison des paramètres actuels aux paramètres formels
2 - évaluation des différentes expressions du corps **<s1>** ... **<sN>**. La valeur retournée par l'appel de la fonction sera la valeur de la dernière évaluation (i.e. celle de **<sN>**)
3 - destruction des liaisons effectuées en 1 -.

Le type de liaison des paramètres est fonction de la forme de la liste des paramètres formels **<lvar>**.
- Si **<lvar>** est une liste de variables la liaison s'effectue élément par élément entre la liste des variables (paramètres formels) et la liste des valeurs (paramètres actuels évalués). Si la liste des paramètres actuels est plus longue que celle des paramètres formels, les paramètres actuels excédentaires seront ignores sans être évalués. Si la liste des paramètres formels est plus longue que celle des paramètres actuels, les variables restantes seront liées à la valeur NIL par défaut, faisant ainsi office de variables locales.
- Si **<lvar>** est une variable simple, tous les paramètres actuels sont évalués puis rassemblés en une liste qui est liée à la variable.
- Si **<lvar>** est une liste pointée de variables, la liaison s'effectue suivant les 2 modes précédents (voir le dernier exemple).

exemples de liaison des EXPR

```
((LAMBDA (X Y Z) (LIST X Y Z))
    (PRIN 1) (PRIN 2) (PRIN 3) (PRIN 4))
1 2 3  et ☞  (1 2 3)

((INTERNAL EXPR ((X Y Z) (LIST X Y Z)))
    (PRIN 1))
1  et ☞  (1 NIL NIL)

((LAMBDA X X) (PRIN 1) (PRIN 2) (PRIN 3))
1 2 3  et ☞  (1 2 3)

((INTERNAL EXPR ((X Y . Z) (LIST X Y Z)))
    (PRIN 1) (PRIN 2) (PRIN 3) (PRIN 4))
1 2 3 4  et ☞  (1 2 (3 4))
```

2.3.4 Les FEXPR

Les FEXPR sont des fonctions écrites en VLISP et interprétées par les fonctions standards d'évaluation (EVAL et APPLY). Tout comme les EXPR ces fonctions possèdent un nombre quelconque de paramètres formels, qui sont rangés dans une liste **<lvar>** ainsi qu'un corps de fonction constitué d'un nombre quelconque d'expressions **<s1> ... <sN>**.

On décrit anonymement une fonction de ce type en utilisant une liste de la forme :

(INTERNAL FEXPR (<lvar> <s1> ... <sN>))

dans laquelle **<lvar>** est la liste des paramètres formels et **<s1> ... <sN>** le corps de la fonction. Ces fonctions ne diffèrent des EXPR que par le mode de liaison des paramètres. Tous les paramètres actuels (i.e. la CDR de la forme elle-même) sont rassemblés sans être évalués en une liste qui est liée à la première variable de la liste des paramètres formels. Toutes les autres variables de cette liste font office de variables locales et sont liées à la valeur NIL.

exemple de liaison des FEXPR

```
((INTERNAL FEXPR ((X) X) (PRINT (CAR '(A B)))))
☞ ((PRINT (CAR '(A B))))

((INTERNAL FEXPR ((X Y . Z) (LIST X Y Z)))
   (PRIN 1) (PRIN 2) (PRIN 3) (PRIN 4))
☞ (((PRIN 1) (PRIN 2) (PRIN 3) (PRIN 4)) NIL NIL)
```

2.3.5 Les MACRO

L'interprète reconnait un autre type de fonctions, les MACRO. Tout comme les EXPR et les FEXPR, ces fonctions écrites en VLISP possèdent un nombre quelconque de paramètres formels, qui sont rangés dans une liste **<lvar>** ainsi qu'un corps de fonction constitué d'un nombre quelconque d'expressions **<s1> ... <sN>**.

On décrit anonymement une fonction de ce type en utilisant une liste de la forme :

(INTERNAL MACRO (<lvar> <s1> ... <sN>))

dans laquelle **<lvar>** est la liste des paramètres formels et **<s1> ... <sN>** le corps de la fonction. Pour évaluer une forme dont la fonction est une MACRO, l'interprète évalue tout d'abord la fonction associée à cette MACRO avec toute la forme comme paramètre actuel puis re-évalue la valeur retournée de cette première évaluation. L'évaluation d'une macro se fait donc en deux temps.
C'est l'appel de la macro tout entier qui est passé en argument et qui est lié à la première variable de la liste des paramètres formels (les autres variables servant de variables locales au corps de la MACRO), il est donc possible de réaliser des modifications physiques de la forme elle-même à la première évaluation de la macro (voir les exemples de la section suivante).

exemple d'appel d'une macro

```
((INTERNAL MACRO ((l) (CADR l)) (ADD1 4)))  ☞ 5
```

2.4 DEFINITION DES FONCTIONS

Tout comme les variables, il existe deux types d'utilisation des fonctions :

- une utilisation statique : les fonctions sont définies d'une manière globale et gardent leurs définitions tant qu'on ne les modifient pas explicitement. Ce type de définition ne permet pas de conserver les définitions antérieures.
- une utilisation dynamique : les fonctions peuvent changer de définition durant certaines évaluations et retrouver par la suite leurs anciennes définitions.

Ces deux types d'utilisation s'appliquent aussi bien aux fonctions nommées (associées à un atome littéral) qu'aux fonctions anonymes (les formes INTERNAL ou les lambda-expressions), de n'importe quel type.

Il faut se rappeler que les définitions des fonctions sont stockées dans les propriétés naturelles F-VAL et F-TYPE des atomes littéraux et qu'il est toujours possible de manipuler ces attributs directement au moyen des fonctions FTYPE et FVAL. Toutefois un certain nombre de fonctions standards permettent de réaliser des définitions statiques ou dynamiques à peu de frais (voir le paragraphe : définition des fonctions du chapitre suivant).





2.5 L'EVALUATEUR

L'évaluateur VLISP 8 (i.e. les fonctions interprète EVAL et APPLY) a été spécialement étudié du point de vue de la vitesse d'exécution.

Toutes les fonctions atomiques sont lancées très rapidement au moyen d'un branchement indirect indexé (par la F-VAL et le F-TYPE).

L'évaluateur minimise le nombre de CONS internes, en n'utilisant jamais la fonction EVLIS pour ses besoins propres (même pour évaluer les arguments des lambda-expressions) sauf en cas de demande explicite au moyen d'une fonction de type LEXPR.

Une particularité maîtresse de VLISP, quelque soit l'implémentation, est d'interpréter itérativement les fausses récursivités (1). Itérativement est ici défini en termes de ressources : un appel de fonction est itératif s'il ne demande pas plus de ressources que celles accordées à l'entrée de la fonction. Les ressources en question sont les tailles de piles, ainsi que le nombre de doublets de listes.

C'est ainsi que dans :

```
(DE LOOPEVAL (IT)
    (LOOPEVAL (PRINT (EVAL (READ)))))
```

la suite des appels internes de LOOPEVAL ne provoquera PAS un débordement de pile, et bouclera, comme il se doit pour une boucle-système, indéfiniment.

Cette propriété se conserve, quelque soit le niveau d'imbrication des appels dits itératifs dans les structures de contrôle mises en jeu dans le corps de fonction.

Toutefois le système ne traite pas les cas de CO-POST-RECURSION.

L'utilisation systématique de cette propriété induit un style très souple et très naturel de programmation récursive, bien éloigné des horribles structures de contrôle dites plates.

---

(1) voir ***Contribution à la définition interprétative et à l'implémentation des lambda-langages,*** par Patrick GREUSSAY, Thèse, Laboratoire Informatique Théorique et Programmation, Novembre 1977, No 78-2, 2 place Jussieu, 75221 Paris Cedex 05, FRANCE.

2.6 DEFINITION META-CIRCULAIRE DE L'INTERPRETE

Enfin voici une description méta-circulaire de l'interprète VLISP. Cette description permet d'avoir une vue globale du fonctionnement de l'interprète mais ne représente pas la véritable mise en oeuvre qui est beaucoup plus performante notamment au niveau du nombre de CONS réalisés dans les fonctions de l'interprète et au niveau de l'encombrement de la pile : l'interprète VLISP ne réalise ***aucun*** CONS pour ses besoins propres.

```
(DE TOP-LEVEL ()
   ; boucle principale ;
   (PRINT "Toplevel")
   (SETQ IT (PRINT (ESCAPE ERREUR (LET (stack) (EVAL (READ))))))
   (PRINT ">" IT)
   (TOP-LEVEL))


(DE EVAL (forme)
   ; évaluation d'une forme quelconque
   (COND
      ((NUMBP forme) forme)
      ((LITATOM forme)
         (IF (BOUNDP forme)
            (CVAL forme)
            (ERREUR (LIST "Atome indéfini : " forme))))
      (T (LET ((fonct (CAR forme)) (larg (CDR forme)))
           (COND
              ((NUMBP fonct) (CNTH fonct (EVAL (CAR larg))))
              ((LITATOM fonct)
                  (EVALINTERNAL (FTYPE fonct) (FVAL fonct) larg))
              ((EQ (CAR fonct) 'LAMBDA)
                  (EVALINTERNAL 'EXPR (CDR fonct) larg))
              ((EQ (CAR fonct) 'INTERNAL)
                  (EVALINTERNAL (CADR fonct) (CDDR fonct) larg))
              (T (SELF (EVAL fonct) larg)))))))


(DE EVALINTERNAL (ftype fval larg)
   ; évaluation d'une fonction suivant son type
   (SELECTQ ftype
      (0SUBR (CALL fval))
      (1SUBR (CALL fval (EVAL (CAR larg))))
      (2SUBR (CALL fval (EVAL (CAR larg)) (EVAL (CADR larg))))
      (3SUBR (CALL fval (EVAL (CAR larg))
                        (EVAL (CADR larg)) (EVAL (CADDR larg))))
      (NSUBR (CALL fval (EVLIS larg)))
      (FSUBR (CALL fval larg))
      (EXPR (EVALAMBDA fval (EVLIS larg)))
      (FEXPR (EVALAMBDA fval (LIST larg)))
      (MACRO (EVAL (EVALAMBDA fval (LIST forme))))
      (T (ERREUR (LIST "EVAL : Fonction indéfinie : " fonct)))))


(DE EVLIS (l)
   ; construit la liste des valeurs des évaluations
   ; de tous les éléments de l
   (IF (NULL l)
      ()
      (CONS (EVAL (CAR l)) (EVLIS (CDR l)))))
```

```
(DE EPROGN (l)
   ; évalue le corps l
   (IF (NULL (CDR l))
      (EVAL (CAR l))
      (EVAL (CAR l))
      (EPROGN (CDR l))))

(DE EVALAMBDA (f larg ;; lvar tok)
   ; applique une F-VAL de type : (<lvar> <s1> ... <sN>)
   ; à la liste d'arguments larg ;
   (SETQ lvar (CAR f))
   (PUSH '**Marker**)
   (WHILE (LISTP lvar)
      (PUSH (CAR lvar) (CVAL (CAR lvar)))
      (SET (NEXTL lvar) (NEXTL larg)))
   (AND lvar
      (PROGN (PUSH lvar (CVAL lvar)) (SET lvar larg)))
   (PROG1
      (EPROGN (CDR f))
      (UNTIL (EQ (SETQ tok (POP)) '**Marker**)
         (SET (POP) tok))))


; avec les fonctions de manipulation de pile

(DE PUSH l (WHILE l (NEXTL stack (NEXTL l))))

(DE POP () (NEXTL stack))


; et pour les fanatiques, un véritable APPLY

(DE APPLY (fonct larg)
   ; applique la fonction fonct aux arguments larg
   (COND
      ((NUMBP fonct)
         (CNTH fonct (CAR larg)))
      ((LITATOM fonct)
         (APPLYINTERNAL (FTYPE fonct) (FVAL fonct) larg))
      ((EQ (CAR fonct) 'LAMBDA)
         (APPLYINTERNAL 'EXPR fonct larg))
      ((EQ (CAR fonct) 'INTERNAL)
         (APPLYINTERNAL (CADR fonct) (CDDR fonct) larg))
      (T (APPLY (EVAL fonct) larg))))


(DE APPLYINTERNAL (ftyp fval larg)
   ; application d'une fonction suivant son type
   (SELECTQ ftyp
      (0SUBR (CALL fval))
      (1SUBR (CALL fval (CAR larg)))
      (2SUBR (CALL fval (CAR larg) (CADR larg)))
      (3SUBR (CALL fval (CAR larg) (CADR larg) (CADDR larg)))
      (NSUBR (CALL fval larg))
      (FSUBR (CALL fval larg))
      (EXPR (EVALAMBDA fval larg))
      (FEXPR (EVALAMBDA fval (LIST larg)))
      (T (ERREUR (LIST "APPLY : Fonction indéfinie." fonct)))))
```





III - LES FONCTIONS STANDARDS

Toutes les fonctions qui vont être décrites dans ce chapitre sont résidentes dans les systèmes actuels.
Pour chacune d'elles on donnera son type (SUBR ou FSUBR) ainsi que le nombre standard d'arguments, et pour chaque argument le type souhaité, ces types étant notés :

<s> pour une S-expression quelconque
<l> pour une liste
<a> pour un atome quelconque (atome littéral ou nombre)
<at> pour un atome littéral
<n> pour un nombre
<c> pour un caractère (i.e. un atome dont le P-NAME n'a qu'un caractère)
<fn> pour une fonction (le nom d'un atome, une lambda-expression ou une fonction explicite de type INTERNAL)

Dans la mesure du possible, les SUBR seront décrites en VLISP sous forme de DE, DF ou DM. Ces descriptions ne sont que l'équivalent VLISP de ces fonctions et ne représenteront que leurs caractéristiques fondamentales.

Dans ce chapitre les fonctions standards sont ordonnées par grandes familles et de nombreuses références avants apparaissent inévitablement durant les définitions de ces fonctions.

ATTENTION : du fait de la limitation chronique en place mémoire de certains systèmes VLISP 8, certaines fonctions (peu usitées) décrites dans ce chapitre sous forme de SUBR ou de FSUBR n'y existent pas sous cette forme. Pour pouvoir les utiliser, il faut obligatoirement les définir sous forme de EXPR, de FEXPR ou de MACRO. En cas de doute sur l'existence d'une fonction, testez la, au toplevel, au moyen d'un exemple simple.

3.1 LES FONCTIONS D'EVALUATION

(EVAL <s>) [SUBR à 1 argument]

C'est la fonction principale de l'interprète. EVAL retourne la valeur de l'évaluation de l'argument **<s>** (voir la description de cette fonction dans le chapitre précédent).

```
ex : (EVAL '(1+ 55))                   ☞   56
     (EVAL (LIST 'ADD 8 '(1+ 3)))      ☞   12
     (EVAL '(2 '(A B C))               ☞   B
     (EVAL '((CAR '(CDR)) '(A B C)))   ☞  (B C)
```

(EVLIS <l>) [SUBR à 1 argument]

retourne une liste composée des valeurs des évaluations de tous les éléments de la liste **<l>**.

Cette fonction peut être décrite en VLISP :

```
(DE EVLIS (l)
   (IF (NULL l)
       ()
       (CONS (EVAL (CAR l)) (EVLIS (CDR l)))))
```

```
ex : (SETQ L '((ADD1 5) (ADD1 7) (ADD1 9)))
      ☞    ((ADD1 5) (ADD1 7) (ADD1 9))
     (EVLIS L)    ☞    (6 8 10)
```

(EPROGN <l>) [SUBR à 1 argument]

évalue séquentiellement tous les éléments de la liste **<l>**. EPROGN retourne la valeur de la dernière évaluation (i.e. celle du dernier élément de **<l>**).

Cette fonction peut être décrite en VLISP :

```
(DE EPROGN (l)
   (IF (NULL (CDR l))
       (EVAL (CAR l))
       (EVAL (CAR l))
       (EPROGN (CDR  l))))
```

```
ex : (SETQ L '((PRIN 1)(PRIN 2)(PRIN 3)))
      ☞    ((PRIN 1)(PRIN 2)(PRIN 3))
     (EPROGN L)  1 2 3  ☞  3
```

(PROG1 <s1> ... <sN>) [FSUBR]

évalue séquentiellement les différentes expressions **<s1>** ... **<sN>**. PROG1 retourne la valeur de la première évaluation (i.e. celle de **<s1>**).

PROG1 peut être décrite en VLISP de la manière suivante :

```
(DF PROG1 (l ;; result)
   (SETQ result (EVAL (CAR l)))
   (EPROGN (CDR l))
   result))
```

ou bien sous forme d'EXPR :

```
(DE PROG1 l (CAR l))
```

ex : (PROG1 (PRIN 1)(PRIN 2)(PRIN 3)) 1 2 3 ☞ 1

PROG1 est souvent utilisé pour réaliser des effets de bord. Voici une MACRO (EXCH var1 var2) qui réalise l'échange des valeurs des variables var1 et var2 sans utiliser de mémoire auxilliaires :

```
(DM EXCH (l)
   (RPLACB l
      (LIST 'SETQ (CADR l)
         (LIST 'PROG1 (CADDR l)
            (LIST 'SETQ (CADDR l) (CADR l))))))
```

avec cette MACRO un appel de type : (EXCH var1 var2)

est transformé en : (SETQ var1 (PROG1 var2 (SETQ var2 var1)))

(PROGN <s1> ... <sN>) [FSUBR]

évalue en séquence les différentes expressions **<s1> ... <sN>**. PROGN retourne la valeur de la dernière évaluation (i.e. celle de **<sN>**).

PROGN est la forme FSUBRée de la fonction EPROGN et peut donc être décrite sous forme de FEXPR de la manière suivante :

```
(DF PROGN (l) (EPROGN l))
```

ou bien sous forme d'EXPR :

```
(DE PROGN l (CAR (LAST l)))
```

ex : (PROGN (PRIN 1)(PRIN 2)(PRIN 3)) 1 2 3 ☞ 3

(QUOTE <s>) [FSUBR]

retourne la S-expression **<s>** non-évaluée. Cette fonction est utilisée comme argument des fonctions de type SUBR ou EXPR dont on ne désire pas évaluer les arguments.
Il existe un macro-caractère standard qui facilite cette écriture, la caractère quote (l'apostrophe) ' . Ce caractère placé devant une expression quelconque **<s>** fabriquera à la lecture la liste **(QUOTE <s>)**.

QUOTE peut être défini en VLISP de la manière suivante :

```
(DF QUOTE (L) (CAR L))
```

```
ex : (QUOTE (ADD1 4))   ☞  (ADD1 4)
     '(A (B C))         ☞  (A (B C))
     'A                 ☞  A
     ''A                ☞  (QUOTE A)
     '''A               ☞  (QUOTE (QUOTE A))
```

3.2 LES FONCTIONS D'APPLICATION

Dans certaines ***petites*** versions du systèmes VLISP 8 (comme les systèmes TRS80(K7) et SDK80), les arguments de type **<fn>** de ces fonctions ne peuvent être que de type lambda-expression.

(APPLY <fn> <l>) [SUBR à 2 arguments]

retourne la valeur de l'application de la fonction **<fn>** à la liste d'arguments **<l>**. La fonction **<fn>** doit être une fonction statique de n'importe quel type SUBR FSUBR EXPR FEXPR ou MACRO.

ex : (APPLY (LAMBDA (x y) (+ x y)) (LIST (1+ 8) (\ 10 3))) ☞ 10

(APPLYN <fn> <s1> ... <sN>) [SUBR à N arguments]

est équivalente à la fonction APPLY. Toutefois les arguments de la fonction ne sont pas regroupés sous la forme d'une liste mais sont arguments de la fonction APPLYN. APPLYN est donc une autre forme de la fonction APPLY.
L'appel (APPLYN <fn> <s1> ... <sN>) est équivalent
à l'appel (APPLY <fn> (LIST <s1> ... <sN>)).

APPLYN peut être défini en VLISP de la manière suivante :

```
(DE APPLYN l
   (APPLY (CAR l) (CDR l)))
```

ex : (APPLYN (LAMBDA (x y) (CONS x y)) 'A 'B) ☞ (A . B)

(MAP <fn> <l>) [SUBR à 2 arguments]

applique successivement la fonction **<fn>** à la liste **<l>** puis à tous ses CDR. MAP retourne NIL en valeur.

MAP peut être défini en VLISP de la manière suivante :

```
(DE MAP (fn l)
   (IF (NULL l)
      ()
      (APPLY fn (LIST l))
      (MAP fn (CDR l))))
```

```
ex : (MAP  (LAMBDA (x) (PRINT x)) '(A (B C) D))
     (A (B C) D)
     ((B C) D)
     (D)
     ☞ NIL
```

(MAPLIST <fn> <l>) [SUBR à 2 arguments]

applique successivement la fonction **<fn>** à la liste **<l>** puis à tous ses CDR. MAPLIST retourne en valeur la liste de toutes les applications successives.

MAPLIST peut être défini en VLISP de la manière suivante :

```
(DE MAPLIST (fn l)
   (IF (NULL l)
      ()
      (CONS (APPLY fn (LIST l)) (MAPLIST fn (CDR l)))))
```

ex :
```
(MAPLIST  (LAMBDA (x) (PRINT x)) '(A (B C) D))
(A (B C) D)
((B C) D)
(D)
☞  ((A (B C) D) ((B C) D) (D))
```

(MAPC <fn> <l>) [SUBR à 2 arguments]

applique successivement la fonction **<fn>** avec tous les CAR de la liste **<l>** comme argument. MAPC retourne NIL en valeur.

MAPC peut être défini en VLISP de la manière suivante :

```
(DE MAPC (fn l)
   (IF (NULL l)
      ()
      (APPLY fn (LIST (CAR l)))
      (MAPC fn (CDR l))))
```

ex:
```
(MAPC (LAMBDA (x) (PRINT x)) '(A (B C) D))
A
(B C)
D
☞ NIL
```

(MAPCAR <fn> <l>) [SUBR à 2 arguments]

applique successivement la fonction **<fn>** avec tous les CAR de la liste **<l>** comme argument. MAPC retourne en valeur la liste des applications successives.

MAPCAR peut être défini en VLISP de la manière suivante :

```
(DE MAPCAR (fn l)
   (IF (NULL l)
      ()
      (CONS (APPLY fn (LIST (CAR l))) (MAPCAR fn (CDR l)))))
```

ex:
```
(MAPCAR (LAMBDA (x) (PRINT x)) '(A (B C) D))
A
(B C)
D
☞  (A (B C) D)
```

(LAMBDA <l> <s1> ... <sN>) [FSUBR]

la valeur d'une lambda-expression est cette lambda-expression elle-même. Cette nouvelle fonction a été rajoutée pour éviter de "quoter" les lambda-expressions explicites des fonctions d'application.

LAMBDA peut être défini en VLISP sous forme d'une FEXPR :

```
(DF LAMBDA (L) (CONS 'LAMBDA L))
```

La définition sous forme de MACRO :

```
(DM LAMBDA (L) L)
```

ferait boucler la fonction interprète EVAL (voyez-vous pourquoi ?)

```
ex : (LAMBDA (X) X)                            ☞   (LAMBDA (X) X)
     (MAPC (LAMBDA (X) (PRIN X)) '(A B C)) A B C ☞   NIL
```

(INTERNAL <s> <s1> ... <sN>) [FSUBR]

comme la forme LAMBDA la valeur d'une forme INTERNAL est cette forme elle-même. Cette nouvelle fonction a été rajoutée pour éviter de "quoter" les fonctions explicites utilisées dans les fonctions d'application.

INTERNAL peut être défini en VLISP sous forme d'une FEXR :

```
(DF INTERNAL (L) (CONS 'INTERNAL L))
```

```
ex : (INTERNAL 1SUBR -6326)  ☞   (INTERNAL 1SUBR -6326)
     ; si l'adresse de la fonction CDR est -6329
     (APPLY (INTERNAL 1SUBR -6329) '(A B C))  ☞   (B C)
```

(SELF <s1> ... <sN>) [SUBR à N arguments]

retourne la valeur de l'application de la dernière lambda-expression utilisée dynamiquement aux différents arguments **<s1> ... <sN>.** Cette fonction permet de résoudre le problème des anciennes fonctions LABEL sans avoir à nommer obligatoirement les fonctions anonymes récursives. Si SELF est employée dans un mauvais contexte (i.e. à l'extérieur d'une lambda-expression), une erreur apparait dont le libellé est :

***** erreur SELF.*** ***ou***
***** SELF error.***

SELF ne peut pas être défini en VLISP 8.2 car cette fonction doit avoir accès à la pile de contrôle de l'interprète.

```
ex : ((LAMBDA (L)
        (IF (NULL (CDR L))
               (CAR L)
               (SELF (CDR L))))
      '(A B C D))
     ☞  D
```

(ESCAPE <at> <s1> ... <sN>) [FSUBR]

est la fonction de contrôle la plus puissante des systèmes VLISP. **<at>** est un atome littéral qui devient le nom d'une fonction dynamique d'échappement, puis les différentes expressions **<s1>** ... **<sN>** sont évaluées en séquence. Si au cours de ces évaluations et à n'importe quel niveau d'appel, une forme de type **(<at> <ss1> ... <ssN>)** est rencontrée, l'évaluation courante est stopppée, les différentes expressions **<ss1>** ... **<ssN>** sont évaluées en séquence et la valeur de **<ssN>** est retournée directement comme valeur de la fonction ESCAPE (en abandonnant toute autre évaluation en cours). Si une telle forme n'est pas rencontrée, ESCAPE retourne la valeur de l'évaluation de **<sN>**.
A la sortie du ESCAPE l'atome **<at>** ne possède plus la définition de fonction d'échappement qu'il avait donc eu de manière provisoire (durant l'exécution du corps de la fonction ESCAPE).

```
ex : (ESCAPE TERM
        (MAPC (LAMBDA (X) (IF (NUMBP X) (TERM 'OUI)))
            '(A B 2 C 3))
        'NON)
     ☞  OUI
```

Dans cet exemple l'évaluation de (TERM 'OUI) retourne directement la valeur OUI à la fonction ESCAPE sans attendre la fin de l'évaluation du MAPC.

```
     (LIST (FTYPE 'TERM)
           (ESCAPE TERM (FTYPE 'TERM))
           (FTYPE 'TERM))
     ☞  (NIL ESCAPE NIL)
```

(EXIT <s1> ... <sN>) [FSUBR]

évalue les différentes expressions **<s1>** ... **<sN>** en séquence et retourne la valeur de la dernière évaluation **<sN>**. De plus EXIT fait sortir de la dernière fonction appelée dynamiquement (fonction utilisateur de type DE, DF ou DM, fonction anonyme de type LET, LAMBDA ou INTERNAL ou fonction dynamique de type WHERE ou ESCAPE) avec pour valeur la valeur du EXIT.
Si un appel de EXIT ne se trouve pas à l'intérieur d'une fonction une erreur apparait dont le libellé est :

*** erreur EXIT. ou*
*** EXIT error.*

ATTENTION : cette fonction s'appelait LESCAPE dans une version antérieure de VLISP.

```
ex : (LET ((n 5) (l NIL))
        (WHILE T
            (IF (= n 0)
                (EXIT l)
                (SETQ l (CONS (SETQ n (1- n)) l)))))
     ☞  (0 1 2 3 4)
```

3.3 LES FONCTIONS DE CONTROLE

Toutes les fonctions de cette section vont permettre de rompre le déroulement séquentiel des évaluations. C'est la raison pour laquelle elles sont toutes de type FSUBR i.e. que les arguments ne seront pas évalués par EVAL mais pour les fonctions elles-mêmes et cela sélectivement.

La fonction de contrôle la plus simple est la fonction conditionnelle IF. Cette fonction va être utilisée pour décrire en VLISP toutes les autres fonctions de contrôle sous forme de FEXPR. Rappelons qu'à l'appel d'une fonction de type FEXPR la liste des arguments non évalués (i.e. le CDR de l'appel lui-même) est liée à la 1ère variable de la liste des variables et le reste des variables de cette liste sont toutes liées à la valeur NIL.

(IF <s1> <s2> <s3> ... <sN>) [FSUBR]

si la valeur de l'évaluation de **<s1>** est différente de NIL, IF retourne la valeur de l'évaluation de l'expression **<s2>**, sinon IF évalue en séquence les différentes expressions **<s3>** ... **<sN>** et retourne la valeur de la dernière évaluation **<sN>**. IF permet de construire une structure de contrôle de type :

si <s1> alors <s2> sinon <s3> ... <sN>

```
ex : (IF T 1 2 3)     ☞ 1
     (IF NIL 1 2 3)   ☞ 3

     ; voici la fonction d'ACKERMANN décrite avec des IF

     (DE ACK (x y)
        (IF (= x 0)
           (1+ y)
           (ACK (1- x)
              (IF (= y 0)
                 1
                 (ACK x (1- y))))))
```

(IFN <s1> <s2> <s3> ... <sN>) [FSUBR]

si la valeur de l'évaluation de **<s1>** est égale à NIL, IFN retourne la valeur de l'évaluation de l'expression **<s2>**, sinon IFN évalue en séquence les différentes expressions **<s3>** ... **<sN>** et retourne la valeur de la dernière évaluation **<sN>**. IFN permet de construire une structure de contrôle de type :

si non <s1> alors <s2> sinon <s3> ... <sN>

IFN est donc équivalent à (IF (NOT <s1>) <s2> <s3> ... <sN>).

et peut être défini en VLISP de la manière suivante :

```
(DF IFN (l)
   (IF (NOT (EVAL (CAR l)))
      (EVAL (CADR l))
      (EPROGN (CDDR l))))

ex : (IFN T 1 2 3)     ☞ 3
     (IFN NIL 1 2 3)   ☞ 1
```

(OR <s1> ... <sN>) [FSUBR]

évalue successivement les différentes expressions **<s1> ... <sN>** jusqu'à ce que l'une de ces évaluations ait une valeur différente de NIL. OR retourne cette valeur.

OR peut être défini en VLISP de la manière suivante :

```
(DF OR (l ;; resul)
   (LET (l l)
      (IF (NULL (CDR l))
        (EVAL (CAR l))
        (IF (SETQ resul (EVAL (CAR l)))
           resul
           (SELF (CDR l))))))
```

ex : (OR NIL NIL 2 3) ☞ 2

(AND <s1> ... <sN>) [FSUBR]

évalue successivement les différentes expressions **<s1> ... <sN>** jusqu'à ce que la valeur d'une évaluation soit égale à NIL, à ce moment AND retourne NIL sinon AND retourne la valeur de la dernière évaluation **<sN>**. Si aucune expression n'est fournie, cette fonction retourne T. AND permet de construire une structure de contrôle de type :

si <s1> alors si <s2> alors ... <sN>

AND peut être défini en VLISP de la manière suivante :

```
(DF AND (l)
   (IF (NULL l)
      T
      (LET (l l)
         (IF (NULL (CDR l))
            (EVAL (CAR l))
            (IF (EVAL (CAR l))
               (SELF (CDR l))
               NIL))))))
```

```
ex : (AND)           ☞ T
     (AND NIL)       ☞ NIL
     (AND 1 2 3 4)   ☞ 4
     (AND 1 2 () 4)  ☞ NIL
```

(COND <l1> ... <lN>) [FSUBR]

est la fonction conditionnelle la plus générale de VLISP. Les différents arguments **<l1> ... <lN>** sont des listes appelées clauses qui ont la structure suivante :
(<ss> <s1> ... <sN>)
COND va sélecter une seule de ces clauses : celle dont la valeur de l'évaluation de son premier élément **<ss>** est différente de NIL. COND évalue alors en séquence les différentes expressions **<s1> ... <sN>** de la clause sélectée et retourne la valeur de la dernière évaluation **<sN>**. Si la clause sélectée n'a qu'un élément **<ss>**, COND retourne la valeur de l'évaluation de **<ss>** (i.e. la valeur qui a déclenché la

sélection de cette clause). COND permet de construire des structures de contrôle de type :

si ... alors ... sinon si ... alors ... ...

Si aucune clause n'est sélectée, COND retourne NIL. Pour forcer la sélection de la dernière clause, il est d'usage d'utiliser comme sélecteur l'atome T (dont la valeur est toujours différente de NIL).

COND peut être défini en VLISP de la manière suivante :

```
(DF COND (l ;; select)
   (LET (l l)
      (IF (NULL l)
         ()
         (SETQ select (EVAL (CAAR l)))
         (IF select
            (IF (CDAR l)
               (EPROGN (CDAR l))
               select)
            select))))
```

```
donc :                          est équivalent à :
       (COND                                          (COND
         (p1 e11 e12 e13)                                (p1 (PROGN e11 e12 e13))
         (p2 e21 e22)                                    (p2 (PROGN e21 e22))
         (p3)                                            ((SETQ aux p3) aux)
         (p4 e41))                                       (p4 e41))
```

```
ex : (COND (NIL 1 2) (T 3 4 5))   ☞ 5

     (COND ((ATOM X) 'ATOM)
           ((LISTP (CAR X))  'LIST)
           ((NULL (CDR X)) 'NULL)
           (T 'What?!?))
```

(SELECTQ <s> <l1> ... <lN> <lf>) [FSUBR]

comme pour la fonction COND, SELECTQ va sélecter une des clauses **<l1> ... <lN>**. Le sélecteur de ces clauses est la valeur de l'évaluation de **<s>**, la sélection s'effectue par comparaison du sélecteur avec :
- le CAR (non évalué) de la clause si celui-ci est un atome littéral (en utilisant le prédicat EQ).
- les différents éléments du sélecteur si celui-ci est une liste (en utilisant la fonction de recherche MEMBER). Cette dernière possibilité permet donc de sélecter des objets de n'importe quel type et ce en nombre quelconque.

Dès qu'une clause est sélectée, SELECTQ évalue en séquence le reste de la clause et retourne la valeur de la dernière évaluation.
Si aucune des clauses **<l1>** ... **<lN>** n'est sélectée, SELECTQ selecte automatiquement la dernière clause **<lf>** et retourne la valeur de de la dernière évaluation de **<lf>**.
Cette dernière clause peut commencer par l'atome T, ce qui n'influe pas sur l'évaluation de la clause, mais permet d'avoir une écriture analogue à la fonction COND.
SELECTQ permet donc de construire des aiguillages sur valeurs constantes.
Il est possible de mettre l'atome T en position sélecteur de la dernière clause à la manière des COND.

SELECTQ peut être défini en VLISP de la manière suivante :

```
(DF SELECTQ (L)
   (LET ((sel (EVAL (CAR L))) (cl (CDR L)))
      ; sel = le sélecteur évalué, cl = les clauses non évaluées
      (COND
         ((NULL (CDR cl))
            ; sélection de la clause <lf>
            (EPROGN (CAR cl)))
         ((AND (LITATOM (CAAR cl)) (EQ sel (CAAR cl)))
            ; sélection simple
            (EPROGN (CDAR cl)))
         ((AND (LISTP (CAAR cl)) (MEMBER sel (CAAR cl)))
            ; sélection générale
            (EPROGN (CDAR cl)))
         (T (SELF sel (CDR cl))))))
```

ex :
```
(SELECTQ 'ROUGE
   (VERT 'ESPOIR)
   (ROUGE 'OK)
   (T 'NON))
☞ OK

(SELECTQ 'ROUGE
   ((BLEU VERT ROUGE) 'COULEUR)
   ((ROSE IRIS) 'FLEUR)
   (T 'SAIS-PAS))
☞ COULEUR
```

(WHILE <s> <s1> ... <sN>) [FSUBR]

tant que la valeur de l'évaluation de **<s>** est différente de NIL, WHILE va évaluer en séquence les différentes expressions **<s1> ... <sN>**. WHILE retourne toujours NIL en valeur (qui est la dernière évaluation de **<s>** qui fait sortir de la boucle WHILE). Cette fonction permet de construire des boucles conditionnelles d'une manière fort commode, ainsi que des boucles infinies en utilisant la forme : (WHILE T ...) car l'atome T possède toujours une valeur différente de NIL.

WHILE est équivalent à la forme VLISP :

```
((LAMBDA () (IFN <s> NIL <s1> ... <sN> (SELF))) ())
```

ou bien

```
(LET () (IFN <s> NIL <s1> ... <sN> (SELF)))
```

WHILE peut être défini en VLISP de la manière suivante :

```
(DF WHILE (l)
   (LET ()
      (IF (NULL (EVAL (CAR l)))
         ()
         (EPROGN (CDR l))
         (SELF))))
```

ex :
```
(SETQ S '(A B C D))                       ☞ (A B C D)
(WHILE S (PRIN (NEXTL S))) A B C D        ☞ NIL
```

(UNTIL <s> <s1> ... <sN>) [FSUBR]

tant que la valeur de l'évaluation de **<s>** est égale à NIL, UNTIL va évaluer en séquence les différentes expressions **<s1> ... <sN>**. UNTIL retourne la valeur de la 1ère évaluation de **<s>** différente de NIL (celle qui fait sortir de la boucle UNTIL). Comme la fonction précédente, UNTIL permet de construire des boucles conditionnelles d'une manière fort commode, ainsi que des boucles infinies en utilisant la forme : (UNTIL () ...).

UNTIL est équivalent à la forme VLISP :

```
((LAMBDA () (OR <s> (PROGN <s1> ... <sN> (SELF)))))
```

ou bien (à la valeur retournée près) :

```
(WHILE (NOT <s>) <s1> ... <sN>)
```

UNTIL peut être défini en VLISP de la manière suivante :

```
(DF UNTIL (l)
   (LET ()
      (OR (EVAL (CAR l))
         (PROGN (EPROGN (CDR l)) (SELF)))))
```

```
ex : (SETQ S '(A B C D))                          ☞ (A B C D)
     (UNTIL (NULL S) (PRIN (NEXTL S))) A B C D    ☞ NIL
```

(REPEAT <n> <s1> ... <sN>) [FSUBR]

évalue l'expression **<n>**. Cette valeur doit être un nombre **<n>** REPEAT évalue alors **<n>** fois les différentes expressions **<s1> ... <sN>**. Si l'évaluation de **<n>** n'est pas un nombre strictement positif, la boucle n'est pas exécutée. Dans tous les cas REPEAT retourne NIL.

REPEAT peut être défini en VLISP de la manière suivante :

```
(DF REPEAT (l)
   (LET ((n (EVAL (CAR l))) (l (CDR l)))
      (IF (<= n 0)
         ()
         (EPROGN l)
         (SELF (1- n) l))))
```

```
ex : (REPEAT 10 (PRIN '*)) * * * * * * * * * *   ☞ NIL
     (REPEAT 0 (PRIN '*))  ☞ NIL
```

3.4 LES PREDICATS DE BASE

En VLISP (comme dans la plupart des LISP modernes) la valeur Booléene "fausse" est assimilée à la valeur NIL et la valeur booléenne "vraie" est assimilée à toute valeur différente de NIL (ce qui laisse un très grand choix de représentation ...).
Certains prédicats devant retourner une valeur booléenne "vraie" utiliseront l'atome spécial T (initiale du TRUE anglais) qui par définition possède une valeur différente de NIL (la valeur de l'atome T est l'atome T lui-même).
Cette section ne décrit que les prédicats de base. Les tests de l'arithmétique entière et mixte sont décrits aux sections 3.15 et 3.16.

(ATOM <s>) [SUBR à 1 argument]

teste si **<s>** est un atome, i.e. un atome littéral ou un nombre. ATOM retourne T si ce test est vérifié, NIL dans le cas contraire.

```
ex : (ATOM 'ARGH)      ☞  T
     (ATOM "OUPS")     ☞  T
     (ATOM 42)         ☞  T
     (ATOM '(A B))     ☞  NIL
```

(LITATOM <s>) [SUBR à 1 argument]

teste si **<s>** est un atome littéral. LITATOM retourne T si le test est vérifié et NIL dans le cas contraire.

```
ex : (LITATOM 'ARGH)      ☞  T
     (LITATOM "OUPS")     ☞  T
     (LITATOM 44)         ☞  NIL
     (LITATOM '(A B))     ☞  NIL
```

(NUMBP <s>) [SUBR à 1 argument]

teste si **<s>** est un nombre. NUMBP retourne le nombre **<s>** si le test est vérifié et NIL dans le cas contraire.

```
ex : (NUMBP 'ARGH)      ☞  NIL
     (NUMBP "OUPS")     ☞  NIL
     (NUMBP 44)         ☞  44
     (NUMBP '(A B))     ☞  NIL
```

(LISTP <s>) [SUBR à 1 argument]

teste si **<s>** est une liste. LISTP retourne la liste **<s>** si le test est vérifié et NIL dans le cas contraire.

```
ex : (LISTP 'ARGH)      ☞  NIL
     (LISTP "OUPS")     ☞  NIL
     (LISTP 44)         ☞  NIL
     (LISTP '(A B))     ☞  (A B)
```

(EQ <s1> <s2>) [SUBR à 2 arguments]

sert à tester 2 atomes littéraux. EQ retourne T si les 2 atomes littéraux **<s1>** et **<s2>** sont égaux et NIL s'ils ne le sont pas. Dans le cas où les arguments ne seraient pas des atomes littéraux, la fonction EQ va tester l'égalité des représentations internes des arguments **<s1>** et **<s2>** (EQ teste donc si les deux arguments ont la même ***adresse physique***).
La représentation interne des nombres variant en fonction des différents systèmes, il est recommandé d'utiliser la fonction = (qui est décrite avec les prédicats numériques) pour tester l'égalité des valeurs numériques).

```
ex : (EQ 'A (CAR '(A)))       ☞ T
     (EQ "STRR" "STRR")       ☞ T
     (EQ (ADD1 119) 120)      ☞ T ou NIL en fonction des systèmes
     (EQ '(A B) '(A B))       ☞ NIL
     (SETQ L '(x y))          ☞ (x y)
     (EQ L L)                 ☞ T
```

(NEQ <s1> <s2>) [SUBR à 2 arguments] ***{pour Not EQ}***

est équivalent à (NOT (EQ **<s1>** **<s2>**)).

NEQ peut être défini en VLISP de la manière suivante :

```
(DE NEQ (s1 s2)
   (IF (EQ s1 s2) NIL T))
```

(EQUAL <s1> <s2>) [SUBR à 2 arguments]

est la fonction de comparaison la plus générale et doit être utilisée dès que le type des objets à comparer n'est pas précisément connu. Si **<s1>** et **<s2>** sont des atomes littéraux, EQUAL est identique à la fonction EQ. Si **<s1>** et **<s2>** sont des nombres, EQUAL est identique à la fonction = sinon si **<s1>** et **<s2>** sont des listes, EQUAL teste si elles possèdent la même structure (i.e. si les listes possèdent les mêmes éléments). Si le test est verifié, EQUAL retourne cette structure dans le cas contraire EQUAL retourne NIL.

EQUAL peut être défini en VLISP de la manière suivante :

```
(DE EQUAL (l1 l2)
   (COND ((LITATOM l1)
            (IF (LITATOM l2) (EQ l1 l2) ()))
         ((NUMBP l1)
            (IF (NUMBP l2) (= l1 l2) ()))
         ((ATOM l2) ())
         ((AND (EQUAL (CAR l1) (CAR l2))
               (EQUAL (CDR l1) (CDR l2)))
            l1)))
```

```
ex : (EQUAL "Foo bar" "Foo bar")           ☞ T
     (EQUAL 1214 (1+ 1213))                ☞ 1214
     (EQUAL '(A (B . C) D) '(A (B . C) D)) ☞ (A (B . C) D)
     (EQUAL '(A B C) '(A B C D))           ☞ NIL
```

(NEQUAL <s1> <s2>) [SUBR à 2 arguments] ***{pour Not EQUAL}***

est équivalent à (NOT (EQUAL **<s1> <s2>**)).

NEQUAL peut être défini en VLISP de la manière suivante :

```
(DE NEQUAL (s1 s2)
   (IF (EQUAL s1 s2) NIL T))
```

(NULL <s>) [SUBR à 1 argument]

teste si **<s>** est égal à NIL. NULL retourne T si le test est vérifié et NIL dans le cas contraire.

NULL peut être défini en VLISP de la manière suivante :

```
(DE NULL (s)
   (EQ s NIL))
```

ex : (NULL NIL) ☞ T
(NULL T) ☞ NIL

(NOT <s>) [SUBR à 1 argument]

effectue l'inversion de la valeur booléenne **<s>**. Du fait de la construction des valeurs booléennes en VLISP, cette fonction est identique à la fonction NULL.

(BOUNDP <at>) [SUBR à 1 argument]

l'argument **<at>** doit être un atome littéral. La fonction BOUNDP teste si la valeur de cet atome littéral est définie ou non (un atome littéral est indéfinie si sa valeur est égale à l'atome UNDEF). Si l'atome est indéfinie, BOUNDP retourne la valeur NIL, sinon si l'atome possède une valeur BOUNDP retourne T. Cette fonction permet donc de tester si une variable possède une valeur sans déclencher l'erreur : ** erreur variable indéfinie :

BOUNDP peut être défini en VLISP de la manière suivante :

```
(DE BOUNDP (at)
   (NEQ (CVAL at) 'UNDEF))
```

ex : (BOUNDP T) ☞ T
(BOUNDP NIL) ☞ T
(BOUNDP 'foofoo) ☞ NIL
- si foofoo n'a pas de valeur -

3.5 LES FONCTIONS DE RECHERCHE

(CVAL <at>) [SUBR à 1 argument]

retourne la valeur de l'atome littéral **<at>**. Cette fonction permet d'accéder directement à la C-VAL d'un atome littéral sans risque de provoquer l'erreur :
 ** erreur variable indéfinie :

Du fait de l'implantation particulière des atomes littéraux en VLISP 8, cette fonction est identique à la fonction CAR mais doit être utilisée de préférence à la fonction CAR par souci de transportabilité des programmes VLISP.

```
ex : (SETQ X '(A B))     ☞  (A B)
     (CVAL 'X)           ☞  (A B)
     X                   ☞  (A B)
     (CVAL 'Buzz)        ☞  UNDEF (si l'atome Buzz est indéfini)
     Buzz
```

*** erreur variable indéfinie : Buzz***

(CAR <s>) [SUBR à 1 argument]

si **<s>** est une liste, retourne son premier élément.
Le CAR d'un nombre est indéterminé.
Si **<s>** est un atome littéral, CAR est équivalent à la fonction précédente CVAL.

```
ex : (CAR '(A B C))      ☞  A
     (SETQ X '(U P))     ☞  (U P)
     (CAR 'X)            ☞  (U P)
     (CAR X)             ☞  U
```

(CDR <s>) [SUBR à 1 argument]

si **<s>** est une liste, retourne cette liste sans son premier élément.
Le CDR d'un nombre est indéterminé.
Du fait de l'implantation particulière de VLISP 8, si **<s>** est un atome littéral, la fonction CDR retourne la P-LIST de cet atome (voir la fonction PLIST).

```
ex : (CDR '(A B C))        ☞  (B C)
     (PUT 'X '(U P) 'I)    ☞  X
     (CDR 'X)              ☞  (I (U P))
```

(C...R <s>) [SUBR à 1 argument]

les 14 combinaisons de CAR et de CDR imbriqués (jusqu'à 3 niveaux) sont disponibles.
(CADR <s>) est équivalent à **(CAR (CDR <s>))**
(CDAAR <s>) est équivalent à **(CDR (CAR (CAR <s>)))**

(MEMQ <at> <l>) [SUBR à 2 arguments]

si l'atome littéral **<at>** est un élément de la liste **<l>**, retourne la partie de la liste **<l>** commençant à l'atome littéral **<at>**, sinon retourne NIL. Cette fonction utilise le prédicat EQ pour tester la présence de l'atome littéral **<at>** dans la liste **<l>**.

MEMQ peut être défini en VLISP de la manière suivante :

```
(DE MEMQ (at l)
   (IF (OR (ATOM l) (EQ at (CAR l)))
       l
       (MEMQ at (CDR l))))
```

```
ex : (MEMQ 'C '(A B C D E))   ☞  (C D E)
     (MEMQ 'Z '(A B C D E))   ☞  NIL
```

(MEMBER <s> <l>) [SUBR à 2 arguments]

si l'expression **<s>** est un élément de la liste **<l>**, retourne la partie de la liste **<l>** commençant à l'élément **<s>**, sinon retourne NIL. Cette fonction utilise le prédicat EQUAL et permet donc de rechercher un élément de n'importe quel type dans une liste.

MEMBER peut être défini en VLISP de la manière suivante :

```
(DE MEMBER (s l)
   (IF (OR (ATOM l) (EQUAL s (CAR l)))
       l
       (MEMBER s (CDR l))))
```

```
ex : (MEMBER 'C '(A B C D E))        ☞  (C D E)
     (MEMBER 'Z '(A B C D E))        ☞  NIL
     (MEMBER '(A B) '(A (A B) C))    ☞  ((A B) C)
```

(NTH <n> <l>) [SUBR à 2 arguments]

retourne la partie de la liste **<l>** commençant à son **<n>**ième élément. Si **<l>** n'est pas une liste ou si (LENGTH **<l>**) est plus petit que **<n>**, NTH retourne NIL. Si **<n>** <= 1, NTH retourne la liste **<l>** en entier.

NTH peut être défini en VLISP de la manière suivante :

```
(DE NTH (n l)
   (IF (<= n 1)
       l
       (NTH (1- n) (CDR l))))
```

```
ex : (NTH 3 '(A B C D E))  ☞  (C D E)
```

(CNTH <n> <l>) [SUBR à 2 arguments] *{pour Car NTH}*

retourne le **<n>**ième élément de la liste **<l>**. CNTH est équivalent à **(CAR (NTH <n> <l>))**.
L'appel de CNTH est implicite en cas de forme dont la fonction est un nombre (cf: l'interprète).

CNTH peut être défini en VLISP de la manière suivante :

```
(DE CNTH (n l)
   (IF (<= n 1)
       (CAR l)
       (CNTH (1- n) (CDR l))))
```

```
ex : (CNTH 3 '(A B C D E F))     ☞  C
     ((1+ 4) '(A B C D E F))     ☞  E
```

(LAST <s>) [SUBR à 1 argument]

si **<s>** est une liste, retourne la liste composée de son dernier élément. si **<s>** n'est pas une liste, retourne **<s>** lui-même.

LAST peut être défini en VLISP de la manière suivante :

```
(DE LAST (s)
   (COND
      ((ATOM s) s)
      ((ATOM (CDR s)) s)
      (T (LAST (CDR s)))))
```

```
ex : (LAST '(A B C D E))     ☞  (E)
     (LAST '(A B C . D))     ☞  (C . D)
     (LAST 120)              ☞  120
```

(LENGTH <s>) [SUBR à 1 argument]

si **<s>** est une liste, retourne le nombre d'éléments de cette liste. Si **<s>** n'est pas une liste (donc si **<s>** est un atome), LENGTH retourne 0.

LENGTH peut être défini en VLISP de la manière suivante :

```
(DE LENGTH (l)
   (IF (ATOM l)
       0
       (1+ (LENGTH (CDR l)))))
```

```
ex : (LENGTH '(A (B C) D E )  ☞  4
     (LENGTH "Esar")          ☞  0
```

3.6 LES FONCTIONS DE CREATION DE LISTES

Toutes les fonctions qui vont être décrites fabriquent de nouvelles listes. La gestion de la mémoire allouée aux listes est dynamique et automatique (voir la section 4.2.2 sur le Garbage-collecting).

ATTENTION : dans certaines versions de VLISP 8, il n'est pas permis de créer des doublets de liste dont le CDR est un nombre entier. Si un tel cas se produit, une erreur fatale apparait dont le libellé est :

***** erreur CDR numérique.*** *ou*
***** numeric CDR.***

(CONS <s1> <s2>) [SUBR à 2 arguments]

construit une liste dont le premier élément est **<s1>** et le reste la liste **<s2>**. Si **<s2>** est un atome, CONS produit la paire pointée

(<s1> . <s2>)

```
ex : (CONS 'A '(B C))         ☞  (A B C)
     (CONS 1 'X)              ☞  (1 . X)
     (SETQ l '(X Y Z))        ☞  (X Y Z)
     (EQUAL (CONS (CAR l) (CDR l))
         l)                   ☞  (X Y Z)
```

```
; Trouvée par D. Goossens,
; voyez-vous ce que construit cette fonction ?

     (DE FOO (l ;; x)
        (IF (ATOM l)
           (CONS l x)
           (FOO (CAR l) (IF (CDR l) (FOO (CDR l) x) x))))
```

exemple d'appel :

```
     (FOO '(A (B . C) ((D))))
```

(MCONS <s1> ... <sN>) [SUBR à N arguments] *{pour Multiple CONS}*

réalise un CONS multiple. L'appel de :
    (MCONS s1 s2 ... sN-1 sN)
équivaut à l'appel
    (CONS s1 (CONS s2 ... (CONS sN-1 sN) ...))

MCONS peut être défini en VLISP :

sous forme d'une FEXPR :

```
     (DF MCONS (l)
        (CONS (EVAL (CAR l))
           (EVAL (IF (NULL (CDDR l))
              (CADR l)
              (CONS 'MCONS (CDR l))))))
```

ou bien sous forme d'une MACRO

```
(DM MCONS (l)
   (LIST 'CONS (CADR l)
      (IF (NULL (CDDDR l))
         (CADDR l)
         (CONS 'MCONS (CDDR l)))))
```

```
ex : (MCONS 'A 'B)       ☞  (A . B)
     (MCONS 'A 'B 'C)    ☞  (A B . C)
```

(LIST <s1> ... <sN>) [SUBR à N arguments]

retourne la liste des valeurs des différentes expressions **<s1> ... <sN>.**
En terme de CONS l'appel **(LIST <s1> <s2> ... <sN-1> <sN>)** est équivalent à
(CONS <s1> (CONS <s2> ... (CONS <sN-1> (CONS <sN> NIL)) ...)).

LIST peut être défini en VLISP :

sous forme d'EXPR :

```
(DE LIST l l)
```

ou sous forme de FEXPR :

```
(DF LIST (l)
  (LET (l l)
     (IF (NULL l)
        ()
        (CONS (EVAL (CAR l))
           (SELF (CDR l))))))
```

ou sous forme de MACRO :

```
(DM LIST (l)
  (IF (NULL (CDR l))
     NIL
     (RPLACB l
        (CONS 'CONS
           (CONS (CADR l)
              (IF (NULL (CDDR L))
                 NIL
                 (CONS (CONS 'LIST (CDDR l)))))))))
```

```
ex : (LIST 'A 'B 'C)   ☞  (A B C)
     (LIST)            ☞  NIL
```

(APPEND <l> <s>) [SUBR à 2 arguments]

retourne la concaténation d'une copie du premier niveau de la liste **<l>** à l'expression **<s>.** Si **<s>** n'est pas fourni, **(APPEND <l>)** retourne simplement une copie du premier niveau de **<l>.**

APPEND peut être défini en VLISP de la manière suivante :

```
(DE APPEND (l s)
```

```
    (IF (NULL l)
        s
        (CONS (CAR l) (APPEND (CDR l) s))))
```

```
ex : (APPEND '(A B C))          ☞  (A B C)
     (APPEND '(A B C) '(X Y))   ☞  (A B C X Y)
     (APPEND () '(X Y))         ☞  (X Y)
```

(REVERSE <s1> <s2>) [SUBR à 2 arguments]

retourne une copie inversée du premier niveau de la liste **<s1>.** Si le deuxième argument est fourni, il est ajouté à la fin de la copie de la première liste inversée.

REVERSE peut être défini en VLISP de la manière suivante :

```
(DE REVERSE (s1 s2)
   (IF (ATOM s1)
       s2
       (REVERSE (CDR s1) (CONS (CAR s1) s2))))
```

```
ex : (REVERSE '(A (B C) D))              ☞  (D (B C) A)
     (REVERSE '((X Y) (U P)) '(G O))     ☞  ((U P) (X Y) G O)
```

On prétend parfois que cette fonction inverse aussi la liste l :

```
(DE REV (l)
   (IF (NULL (CDR l))
       l
       (CONS (CAR (REV (CDR l)))
             (REV (CONS (CAR l)
                  (REV (CDR (REV (CDR l))))))))
```

(COPY <l>) [SUBR à 1 argument]

fabrique une nouvelle copie de toute la liste **<l>.** Pour cette fonction la copie s'effectue à tous les niveaux de l'arborescence.

COPY peut être défini en VLISP de la manière suivante :

```
(DE COPY (s)
   (IF (ATOM s)
       s
       (CONS (COPY (CAR s)) (COPY (CDR s)))))
```

```
ex : (COPY 'A)              ☞  A
     (COPY '(A (B (C (D)))) ☞  (A (B (C (D))))
```

Cette fonction ne traite pas les listes circulaires ou partagées. Pour cela il faut utiliser la fonction :

```
(DE CIRCOPY (l ;; d)
   (LET ((l l) (new))
      (COND
         ((ATOM l) l)
         ((CDR (ASSQ l d)))
         (T (SETQ new (CONS NIL NIL)
                  d (CONS (CONS l new) d))
            (RPLACA new (SELF (CAR l)))
            (RPLACD new (SELF (CDR l)))))))
```

(SUBST <s1> <s2> <l>) [SUBR à 3 arguments]

fabrique une nouvelle copie de toute la liste **<l>** en substituant à chaque occurrence de l'expression **<s2>**, l'expression **<s1>**. Si les deux expressions sont les mêmes pointeurs physiques (i.e. si **(EQ <s1> <s2>)** = T), l'interprète se permet d'appeler la fonction COPY à la place de SUBST, l'exécution de la fonction COPY étant beaucoup plus rapide.

SUBST peut être défini en VLISP de la manière suivante :

```
(DE SUBST (s1 s2 l)
   (IF (EQ s1 s2)
      (COPY l)
      (LET (l l)
         (COND ((EQ l s2) s1)
               ((ATOM l) l)
               (T (CONS (SELF (CAR l))
                        (SELF (CDR l)))))))))
```

```
ex : (SUBST '(X Y Z) 'A  '(A C (D A)))
     ☞  ((X Y Z) C (D (X Y Z)))
```

(OBLIST) [SUBR à 0 argument]

retourne la (très longue) liste de tous les atomes littéraux présents dans le système. Cette liste ne contient pas l'atome UNDEF A l'initialisation du système, cette liste contient le nom de toutes les constantes littérales et de toutes les fonctions standards.

Il n'est pas possible de définir simplement cette fonction en VLISP 8 car il faut accéder aux propriétés naturelles de type A-LINK de chacun des atomes, ce qui n'est réalisable qu'avec les fonctions LOC et MEMORY.

```
ex :  (OBLIST)  ☞

      (NIL 0SUBR 1SUBR 2SUBR 3SUBR NSUBR FSUBR EXPR
      FEXPR MACRO T LAMBDA & QUOTE GC EDITV CLEAR
      DISPLAY POINT WINDOW ↑C PRINT PRIN PRINTLEVEL
      RMARGIN LMARGIN OUTPOS PTYPE EXPLODE TERPRI
      PRINCH OBASE READ PEEKCH READCH TEREAD ASCII
      CASCII TYPECH IMPLODE EVAL APPLY EXIT SELF
                     .....
                     .....
                     .....
      MEMORY CALL EXECUTE STOP)
```

3.7 LES FONCTIONS DE MODIFICATION

Toutes les fonctions qui vont être décrites dans cette section doivent être utilisées conformément au mode d'emploi, pour éviter de placer le système dans un état de confusion dramatique car elles vont permettre de modifier **physiquement** les structures VLISP.
Toutefois la possibilité d'une véritable ***chirurgie*** sur les représentations internes des objets confère à VLISP la puissance des langages machines.

Une erreur se produit en cas de tentative de modification de constante (i.e. un nombre ou une constante littérale de type NIL T ...).
Le libellé de cette erreur est :

***** erreur modification de constante : <cst>   ou***
***** altered constant : <cst>***

dans lequel le nom de la constante modifiée ***<cst>*** est imprimé.

Pour ces fonctions l'argument **<obj>** représente soit un atome littéral soit une liste. En VLISP 8, modifier le CAR d'un atome littéral revient à changer sa C-VAL, modifier son CDR revient à modifier sa P-LIST.

(RPLACA <obj> <s>) [SUBR à 2 arguments] ***{pour RePLACe cAr}***

remplace le CAR de **<obj>** par **<s>**. Retourne le nouvel **<obj>** en valeur.

```
ex : (SETQ X '(A B))            ☞  (A B)
     (RPLACA 'X '(C))           ☞  X
     X                          ☞  (C)
     (RPLACA X 'D)              ☞  (D)
     X                          ☞  (D)
     (RPLACA '(A B C) '(X Y))   ☞  ((X Y) B C)
```

(RPLACD <obj> <s>) [SUBR à 2 arguments] ***{pour RePLACe cDr}***

remplace le CDR de **<obj>** par **<s>**. Retourne le nouvel **<obj>** en valeur.

```
ex : (RPLACD '(A B C) '(X Y Z))  ☞  (A X Y Z)
```

(RPLACB <obj> <l>) [SUBR à 2 arguments] ***{pour RePLACe Both}***

remplace le CAR de **<obj>** par le CAR de **<l>** et le CDR de **<obj>** par le CDR de **<l>**. Cette fonction est souvent utilisée dans des MACRO pour modifier physiquement l'appel de MACRO lui-même.

RPLACB peut être défini en VLISP de la manière suivante :

```
(DE RPLACB (obj l)
   (RPLACA obj (CAR l))
   (RPLACD obj (CDR l))
   obj)
```

```
ex : (SETQ L1 '(A B C))       ☞  (A B C)
     (SETQ L2 L1)             ☞  (A B C)
     (RPLACB L1 '(X Y))       ☞  (X Y)
     L2                       ☞  (X Y)
```

(SET <obj> <s>) [SUBR à 2 arguments]

remplace le CAR de **<obj>** par la valeur **<s>.** SET retourne en valeur **<s>.** A la valeur retournée près, **(SET <obj> <s>)** est équivalent à **(RPLACA <obj> <s>).**

```
ex : (SET '(A B C) '(X Y))   ☞  (X Y)
```

(SETQ <at1> <s1> ... <atN> <sN>) [FSUBR]

<at1> ... <atN> sont des atomes littéraux qui ne sont pas évalués ; **<s1> ... <sN>** sont des expressions quelconques qui seront évaluées. SETQ est la fonction d'affectation la plus utilisée : chaque atome **<ati>** est initialisé avec l'expression correspondante **<si>.** SETQ retourne **<sN>** en valeur.

```
ex : (SETQ L1 '(A B C))      ☞  (A B C)
     (SETQ L2 L1)            ☞  (A B C)
     (SETQ L3 L2 L4 'FOO)    ☞  FOO
     L3                      ☞  (A B C)
```

(NEXTL <at>) [FSUBR] *{pour NEXT List}*

<at> (qui n'est pas évalué) doit être un atome littéral dont la valeur doit être une liste. NEXTL retourne le CAR de cette liste en valeur et donne comme nouvelle valeur de **<at>** le CDR de son ancienne valeur. Cette fonction est très utile pour "avancer dans une liste" qui est la valeur d'un atome.

Cette fonction correspond en VLISP à :

```
   (PROG1 (CAR <at>) (SETQ <at> (CDR <at>)))
```

```
ex : (SETQ A '(X Y Z))    ☞  (X Y Z)
     (NEXTL A)            ☞  X
     A                    ☞  (Y Z)
```

(NEWL <at> <s>) [FSUBR] *{pour NEW List}*

<at> (qui n'est pas évalué) doit être un atome littéral dont la valeur est une liste. NEWL place en tête de cette liste la valeur de **<s>** et retourne en valeur la nouvelle liste formée.
Cette fonction correspond en VLISP à :

```
    (SETQ <at> (CONS <s> <at>)).
```

L'utilisation conjuguée des fonctions NEXTL et NEWL permet de construire très naturellement des piles-listes.

```
ex : (SETQ A '(X Y Z))    ☞  (X Y Z)
     (NEWL A 'W)          ☞  (W X Y Z)
     A                    ☞  (W X Y Z)
```

(NCONC <l1> <l2>) [SUBR à 2 arguments]

concatène physiquement les deux listes **<l1>** et **<l2>** (i.e. place dans le CDR du dernier élément de **<l1>** un pointeur sur la liste **<l2>**). NCONC retourne la nouvelle liste **<l1>** en valeur. Si **<l1>** et **<l2>** sont les mêmes pointeurs physiques, NCONC permet de construire des listes circulaires par forcage dans le dernier CDR de la liste d'un pointeur sur le 1er élément de cette même liste.

NCONC peut être défini en VLISP de la manière suivante :

```
(DE NCONC (l1 l2)
   (IF (LISTP l1)
      (LET (l l1)
         (IF (ATOM (CDR l))
            (RPLACD l l2)
            (SELF (CDR l))))
   l1)
```

```
ex : (SETQ X1 '(A B C) X2 X1)  ☞  (A B C)
     (NCONC X1 '(D E F))       ☞  (A B C D E F)
     X2                        ☞  (A B C D E F)
     (NCONC X1 X1)             ☞  (A B C D E F A B C D E F A B ...
```

(FREVERSE <l> <s>) [SUBR à 2 arguments] ***{pour Fast REVERSE}***

renverse physiquement (et rapidement) la liste **<l>**. Si le deuxième argument **<s>** est fourni, il est ajouté au moyen d'un NCONC physique à la fin de la première liste inversée : l'appel correspond donc à :
(NCONC (FREVERSE <l>) <s>)
Cette fonction doit être manipulée avec précaution car elle modifie physiquement des structures VLISP (en particulier les modifications peuvent être ***boulversantes*** en cas de structures partagées), mais elle est évidement beaucoup plus rapide que la fonction traditionnelle REVERSE.

FREVERSE peut être défini en VLISP de la manière suivante :

```
(DE FREVERSE (l s)
   (IF (LISTP l)
      (FREVERSE (CDR l) (RPLACD l s))
      s))
```

```
ex : (SETQ L1 '(A B C D E)) ☞ (A B C D E)
     (SETQ L2 (CDR L1))     ☞ (B C D E)
     (SETQ L3 (LAST L1))    ☞ (E)
     (FREVERSE L1 '(X Y))   ☞ (E D C B A X Y)
     L1                     ☞ (A X Y)
     L2                     ☞ (B A X Y)
     L3                     ☞ (E D C B A X Y)
```

3.8 LES FONCTIONS SUR LES A-LISTES

En VLISP 8 comme dans tous les VLISP, les A-listes (les listes d'association) sont des **tables** (au sens de SNOBOL 4) qui possèdent la structure suivante :

```
((clé1 . val1) (clé2 . val2)   ...   (cléN . valN))
```

Chaque élément d'une A-liste est un couple constitué d'une clé (le CAR de l'élément de la table) et d'une valeur (le CDR de l'élément). L'accès à une valeur est réalisé au moyen de sa clé.

Pour toutes les fonctions qui vont être décrites, l'argument **<al>** doit être une A-liste, et les clés doivent être des atomes littéraux (car les fonctions d'accès aux A-listes utilisent le prédicat EQ pour des raisons d'efficacité).

(ASSQ <at> <al>) [SUBR à 2 arguments]

retourne l'élément de la A-liste **<al>** dont la clé (le CAR) est égal à l'atome littéral **<at>**, sinon ASSQ retourne NIL.

ASSQ peut être défini en VLISP de la manière suivante :

```
(DE ASSQ (at al)
    (COND ((NULL al) NIL)
          ((EQ (CAAR al) at) (CAR al))
          (T (ASSQ at (CDR al)))))
```

ex : (ASSQ 'B '((A) (B 1) (C D E))) ☞ (B 1)

(CASSQ <at> <al>) [SUBR à 2 arguments]

est identique à ASSQ mais retourne la valeur seule (le CDR) de l'élément dont on précise la clé.

(CASSQ at al) est donc équivalent à (CDR (ASSQ at al))

ATTENTION : on ne peut pas distinguer la valeur NIL associée à un élément de A-liste avec l'absence de cet élément.

ex : (CASSQ 'C '((A) (B 1) (C D E))) ☞ (D E)

Si on veut utiliser des clés plus complexes (de n'importe quel type : nombre ou liste) il faut utiliser la fonction suivante pour accéder aux éléments d'une telle A-liste :

(ASSOC <s> <al>) [EXPR]

cette fonction doit être définie en VLISP 8 de la manière suivante :

```
(DE ASSOC (s al)
   (COND
      ((NULL al) NIL)
      ((EQUAL s (CAAR al)) (CAR al))
      (T (ASSOC s (CDR al)))))
```

Voici 2 fonctions utilisant des A-listes, qui ne sont pas des fonctions standards de VLISP 8 (du fait de son manque de place chronique) mais qui existent dans d'autres systèmes VLISP.

(PAIRLIS <l1> <l2> <al>) [EXPR]

<l1> doit être une liste de clés
<l2> doit être une liste de valeurs
PAIRLIS retourne une nouvelle A-liste formée à partir de la liste des clés **<l1>** et de la liste des valeurs associées **<l2>**. Si le troisième argument **<al>** est fourni, il est ajouté à la fin de la A-liste créée.

Voici la définition de PAIRLIS en VLISP 8 :

```
(DE PAIRLIS (l1 l2 al)
   (IF (NULL l1)
       al
       (CONS (CONS (CAR l1) (CAR l2))
          (PAIRLIS (CDR l1) (CDR l2) al))))
```

```
ex : (PAIRLIS '(YA JESTEM WIELKY)  '(LE MERLE CHANTE) ())
          ☞  ((YA . LE) (JESTEM . MERLE) (WIELKY . CHANTE))
     (PAIRLIS '(X Y Z) '(A (B)) '((A . X) (B . Y)))
        ☞  ((X . A) (Y B) (Z) (A . X) (B . Y))
```

(SUBLIS <al> <s>) [EXPR]

retourne une copie de l'expression **<s>** dans laquelle toutes les occurrences des clés de la A-liste **<al>** ont été remplacées par leurs valeurs associées correspondantes. Cette fonction effectue une copie entière de l'expression **<s>** et utilise la fonction ASSQ.

Voici la définition de la fonction SUBLIS en VLISP 8 :

```
(DE SUBLIS (al s)
   (LET (s s)
      (IF (ATOM s)
         (LET (x (ASSQ s al)) (IF x (CDR x) s))
         (CONS (SELF (CAR s)) (SELF (CDR s))))))
```

```
ex :  (SUBLIS '((A . Z) (B 2 3)) '(A (B A C) D B . B)))
   ☞ (Z ((2 3) Z C) D (2 3) 2 3)
```

3.9 LES FONCTIONS SUR LES F-VAL ET LES F-TYPE

On peut associer à chaque atome littéral une définition de fonction en donnant une valeur aux propriétés naturelles F-TYPE et F-VAL de ces atomes. Les fonctions qui vont être décrites contrôlent la validité de leurs arguments. On ne peut associer de fonctions qu'à des atomes littéraux qui ne sont pas des constantes littérales (comme NIL, T, LAMBDA, FUNCTION, QUOTE, EXPR ...). De plus tous les paramètres formels de ces fonctions doivent être des atomes littéraux également. En cas de mauvaise utilisation de ces fonctions une erreur apparait dont le libellé est :

> ** *erreur mauvaise définition.*   *ou*
> ** *bad definition.*

(FTYPE <at> <ftype>) [SUBR à 2 arguments] *{pour Function TYPE}*

Si le 2ème argument **<ftype>** n'est pas fourni (si **<ftype>**=NIL), cette fonction retourne le F-TYPE de la fonction associée à l'atome littéral **<at>** donné en 1er argument.
Ce F-TYPE peut être l'un des atomes suivants :

- 0SUBR si **<at>** possède une définition de type SUBR sans argument.
- 1SUBR si **<at>** possède une définition de type SUBR à 1 argument.
- 2SUBR si **<at>** possède une définition de type SUBR à 2 arguments.
- 3SUBR si **<at>** possède une définition de type SUBR à 3 arguments.
- NSUBR si **<at>** possède une définition de type SUBR à N arguments.
- FSUBR si **<at>** possède une définition de type FSUBR.
- EXPR si **<at>** possède une définition de type EXPR.
- FEXPR si **<at>** possède une définition de type FEXPR.
- MACRO si **<at>** possède une définition de type MACRO.
- ESCAPE si **<at>** possède une définition de type ESCAPE.
- NIL si **<at>** ne possède pas de définition de fonction.

Si le 2ème argument **<ftype>** est fourni, il doit être un des atomes énumérés ci-dessus et il devient le nouveau F-TYPE de l'atome littéral **<at>**.
Pour détruire un F-TYPE il faut utiliser la forme : (FTYPE <at> 0)
et non : (FTYPE <at>)
qui ne permetrait qu'une consultation du F-TYPE.

```
ex : (FTYPE 'FTYPE)      ☞  2SUBR
     (FTYPE 'COND)       ☞  FSUBR
     (LIST (FTYPE 'PUT) (ESCAPE PUT (FTYPE 'PUT)) (FTYPE 'PUT))
         ☞ (3SUBR ESCAPE 3SUBR)
     (FTYPE 'QUOI)       ☞  NIL
```

(FVAL <at> <s>) [SUBR à 2 arguments] *{pour Function VALue}*

permet de retourner (après modification si **<s>** est fourni) la F-VAL associée à l'atome littéral **<at>** ou bien NIL s'il n'y a pas de fonction associée. La F-VAL d'une fonction de type SUBR est son adresse de lancement, la F-VAL d'une fonction de type EXPR/FEXPR/MACRO est une liste de la forme (<lvar> <s1> ... <sN>), et la F-VAL d'une fonction de type ESCAPE est le nom de cette fonction ESCAPE.

ATTENTION : pour détruire la F-VAL d'un atome littéral il ne faut pas employer la forme (FVAL at) qui ne réalise qu'une consultation de cette F-VAL mais la forme (FVAL at 0) qui force l'adresse 0 dans la

F-VAL.

```
ex :  (DE FOO (N) (* N N))   ☞  FOO
      (FTYPE 'FOO)           ☞  EXPR
      (FVAL 'FOO)            ☞  ((N) * N N)
      (FVAL 'CAR)            ☞  12622
                  (cette adresse est dépendante du système utilisé)
```

Ces 2 dernières fonctions permettent de rendre synonymes statiquement des noms de fonctions par exemple au moyen de cette EXPR :

(SYNONYM <at1> <at2>) [EXPR]

permet de donner à l'atome littéral **<at1>** le type et la valeur de la fonction associée àb l'atome littéral **<at2>**.

SYNONYM doit être défini en VLISP de la manière suivante :

```
(DE SYNONYM (at1 at2)
   (FTYPE at1 (FTYPE at2))
   (FVAL at1 (FVAL at2))
   at1)
```

ex :

```
(SYNONYM 'KONS 'CONS)   ☞  KONS
(KONS 'A 'B)            ☞  (A . B)
```

(GETFN <at>) [SUBR à 1 arguments] *{pour GET FuNction}*

permet de retourner la définition de la fonction de type EXPR/FEXPR/MACRO associée à l'atome littéral **<at>** sous la forme de sa définition (i.e. sous la forme de l'appel d'une des fonctions DE/DF/DM).

GETFN peut être défini en VLISP de la manière suivante :

```
(DE GETFN (at)
   (SELECTQ (FTYPE at)
      (EXPR (MCONS 'DE at (FVAL at)))
      (FEXPR (MCONS 'DF at (FVAL at)))
      (MACRO (MCONS 'DM at (FVAL at)))
      (T ())))
```

```
ex :  (DE BAR (N) (+ N N))   ☞  FOO
      (FTYPE 'BAR)           ☞  EXPR
      (FVAL 'BAR)            ☞  ((N) + N N)
      (GETFN 'BAR)           ☞  (DE BAR (N) (+ N N))
```

3.10 LES FONCTIONS DE DEFINITION DE FONCTIONS

Ces fonctions vont permettre de définir de nouvelles fonctions. Elles testent la validité de leurs arguments :
- les noms des fonctions doivent être des atomes littéraux
- toutes les variables de la liste des paramètres formels doivent également être des atomes littéraux.

Si ce n'est pas le cas, une erreur apparait dont le libellé est :

*** erreur mauvaise définition.* *ou*
*** bad definition*

Il existe 2 types de définition de fonction : les définitions statiques et les définitions dynamiques (voir le chapitre précédent sur le fonctionnement de l'interprète).

3.10.1 Définitions des fonctions statiques

(DE <at> <lvar> <s1> ... <sN>) [FSUBR]

permet de définir statiquement de nouvelles EXPR. **<at>** est le nom de l'atome littéral auquel sera rattaché une fonction équivalente à :

(INTERNAL EXPR (<lvar> <s1> ... <sN>))

DE (prononcez "dé" comme en latin) retourne le nom de la fonction **<at>** en valeur.

DE peut être défini en VLISP de la manière suivante :

```
(DF DE (l)
   (FVAL (CAR l) (CDR l))
   (FTYPE (CAR l) EXPR)
   (CAR l))
```

```
ex : (DE FOO (l) (IF (NULL (CDR l)) l (FOO (CDR l)))  ☞ FOO
     (FVAL 'FOO)  ☞  ((l) (IF (NULL (CDR l)) l (FOO (CDR l))))
     (FTYPE 'FOO)  ☞  EXPR
     (FOO '(A B C))  ☞  (C)
```

(DF <at> <lvar> <s1> ... <sN>) [FSUBR]

permet de définir statiquement de nouvelles FEXPR. **<at>** est le nom de l'atome littéral auquel sera rattaché une fonction équivalente à :

(INTERNAL FEXPR (<lvar> <s1> ... <sN>))

DF retourne en valeur le nom **<at>** de la fonction définie.

```
ex : (DF INCR (L) (SET (CAR L) (1+ (CAAR L))))   ☞  INCR
     (TYPEFN 'INCR)  ☞  FEXPR
     (SETQ nb 7)  ☞  7
     (INCR nb)    ☞  8
```

(DM <at> <lvar> <s1> ... <sN>) [FSUBR]

permet de définir statiquement de nouvelles MACRO. **<at>** est le nom de l'atome littéral auquel sera rattachée une fonction équivalente à :

(INTERNAL MACRO (<lvar> <s1> ... <sN>))

DM retourne en valeur le nom **<at>** de la fonction définie.

```
ex : (DM DECR (l) (LIST 'SETQ (CADR l) (LIST '1- (CADR l))))
         ☞ INCR
     (SETQ N 10)  ☞ 10
     (DECR N)     ☞ 9
```

3.10.2 Définitions des fonctions dynamiques

(WHERE <l> <s1> ... <sN>) [FSUBR]

permet de définir des fonctions dynamiquement (d'une manière analogue à la fonction LET pour les variables).

<l> est une liste de la forme :
(<at> <fval>) ou bien **(<at> <ftype> <fval>)**
<s1> ... **<sN>** est un corps de fonction.

WHERE va lier, le temps de l'évaluation de **<s1>** ... **<sN>**, une nouvelle fonction à l'atome **<at>**. Cette fonction est de type **<ftype>** si celui-ci est précisé (ou bien EXPR en cas d'absence de **<ftype>**) et la valeur de cette fonction est **<fval>** (qui est évaluée). Au sortir du WHERE, l'atome littéral **<at>** reprendra son ancienne définition (s'il en avait une).

Il existe un deuxième type de fonctions dynamiques, les fonctions dynamiques de type ESCAPE. Elles sont décrites au paragraphe 3.2 sur les fonctions d'applications.

```
ex : (WHERE (CAR '((x) (CDR x))) (CAR '(A B C)))  ☞ (B C)
     (CAR '(A B C))                               ☞ A
```

3.10.3 Appel de type LET

(LET <l> <s1> ... <sN>) [MACRO]

permet d'appeler une fonction anonyme de type EXPR (i.e. une lambda-expression).
<l> est une liste de la forme :

() dans le cas où il n'y a pas de variable
(<var> <val>) dans le cas où il n'y a qu'une variable
((<var1> <val1>) ... (<varN> <valN>)) dans le cas de variables multiples.

<s1> ... **<sN>** est un corps de fonction.

La fonction LET va lier dynamiquement et en parallèle toutes les variables **<vari>** aux valeurs **<vali>** puis lancer l'exécution du corps de la fonction **<s1>** ... **<sN>**. Au sortir du corps de cette fonction les variables seront déliées et retrouveront leurs anciennes valeurs. LET permet de définir des variables locales très facilement.

LET est une forme abrégée d'appel de fonctions anonymes de type EXPR.

Ainsi la forme
```
        (LET (<var> <val>) <s1> ... <sN>)
```
correspond à l'appel
```
        ((LAMBDA (<var>) <s1> ... <sN>) <val>)
```
ou bien à l'appel
```
        ((INTERNAL EXPR ((<var>) <s1> ... <sN>)) <val>)
```

et la forme
```
        (LET ((<var1> <val1>) ... (<varN> <valN>)) <s1> ... <sN>)
```
correspond à l'appel
```
        ((LAMBDA (<var1> ... <varN>) <s1> ... <sN>) <val1> ... <valN>)
```
ou bien à l'appel
```
        ((INTERNAL EXPR
               ((<var1> ... <varN>) <s1> ... <sN>)) <val1> ... <valN>)
```

LET est en VLISP 8 . 2 la MACRO suivante :

```
(DM LET (ls)
   (RPLACB ls
      (COND
         ((NULL (CADR ls))
            (CONS (CONS LAMBDA (CONS () (CDDR ls)))))
         ((ATOM (CAADR ls))
            (CONS (CONS LAMBDA (CONS (LIST (CAADR ls)) (CDDR ls)))
                (CDADR ls)))
         (T (CONS  (CONS LAMBDA (CONS (MAPCAR 'CAR (CADR ls)) (CDDR ls)))
                (MAPCAR 'CADR (CADR ls)))))))
```

3.11 LES FONCTIONS SUR LES P-LIST DES ATOMES LITTERAUX

En VLISP 8 comme dans tous les VLISP, les P-LIST (listes de propriétés) sont des listes (constituées d'indicateurs et de valeurs) qui ont la structure suivante :

(indic1 val1 indic2 val2 ... indicN valN)

A chaque indicateur **indici** est associée une valeur **vali** qui est l'élément suivant de la P-LIST. Les recherches sur les P-LIST s'effectuent donc deux éléments par deux éléments.

Chaque atome littéral possède une P-LIST qui lui est propre.

Il existe, en VLISP 8, 5 fonctions de manipulation de ces P-LIST.

Les arguments de ces fonctions sont :

<pl> - un atome littéral dont on veut utiliser la P-LIST . Si **<pl>** est un nombre, une liste ou bien l'atome NIL, une erreur fatale apparait dont le libellé est :

*** erreur mauvaise P-LIST : <pl> ou*
*** Bad P-list : <pl>*

dans lequel le nom de la P-LIST incriminée *<pl>* est imprimé.

<ind> un indicateur. Celui-ci doit être un atome littéral, la recherche des indicateurs utilisant le prédicat EQ.

<pval> peut être n'importe quelle expression.

(PLIST <pl>) [SUBR à 1 argument]

retourne la P-LIST associée à l'atome littéral **<pl>**. Si l'atome littéral **<pl>** ne possède pas de P-LIST, PLIST retourne NIL.

Du fait de l'implémentation particulière des P-LIST en VLISP 8, la fonction PLIST est actuellement équivalente à la fonction CDR. Toutefois il est recommandé d'utiliser la fonction PLIST pour des raisons de transportabilité des programmes VLISP.

PLIST peut être défini en VLISP 8 :

```
(DE PLIST (at) (CDR at))
```

mais ne peut pas en général être défini simplement dans les autres systèmes LISP.

(SETPLIST <pl> <l>) [SUBR à 2 arguments]

force la P-LIST de l'atome **<pl>** avec la liste **<l>**. Cette fonction permet donc d'initialiser une P-LIST. SETPLIST retourne la nouvelle P-LIST (i.e. la valeur de **<l>**) en valeur.

* du fait de la représentation des P-LIST en VLISP 8, SETPLIST peut être défini :

```
(DE SETPLIST (pl l)
   (RPLACD pl l))
```

ex : (SETPLIST 'rose '(nom commun genre féminin))
☞ (nom commun genre féminin)
(PLIST 'rose) ☞ (nom commun genre féminin)

(GET <pl> <ind>) [SUBR à 2 arguments]

retourne la valeur associée à l'indicateur **<ind>** dans la P-LIST **<pl>**. Si l'indicateur n'existe pas, GET retourne NIL.
ATTENTION : on ne peut pas discerner la valeur égale à NIL d'un indicateur avec l'absence de cet indicateur.

GET peut être défini en VLISP de la manière suivante :

```
(DE GET (pl ind)
   (LET (pl (PLIST pl))
      (COND ((NULL pl) NIL)
            ((EQ (CAR pl) ind) (CADR pl))
            (T (SELF (CDDR pl))))))
```

ex : (PLIST 'rose) ☞ (nom commum genre féminin)
(GET 'rose 'genre) ☞ féminin
(GET 'rose 'famille) ☞ NIL

La recherche de plusieurs indicateurs peut être réalisé au moyen de la fonction suivante :

```
(DE GETL (pl l)
   ; pl = une P-LIST
   ; l  = une liste d'indicateurs
   (LET (pl (PLIST pl))
      (COND
         ((NULL pl) pl)
         ((MEMQ (CAR pl) l) pl)
         (T (SELF (CDDR pl))))))
```

Cette fonction retourne la partie de la P-LIST pl à partir d'un des indicateurs contenus dans la liste l.

(ADDPROP <pl> <pval> <ind>) [SUBR à 3 arguments]

rajoute ***en tête*** de la P-LIST **<pl>** l'indicateur **<ind>** et sa valeur associée **<pval>**. ADDPROP retourne **<pl>** en valeur.

ADDPROP peut être défini en VLISP de la manière suivante :

```
(DE ADDPROP (pl pval ind)
   (SETPLIST pl (CONS ind (CONS pval (PLIST pl))))
   pl)
```

ex : (PLIST 'PLT) ☞ (I1 A I2 B)
(ADDPROP 'PLT 'C 'I1) ☞ 'PLT
(PLIST 'PLT) ☞ (I1 C I1 A I2 B)

(PUT <pl> <pval> <ind>) [SUBR à 3 arguments]

si l'indicateur **<ind>** existe déjà sur la P-LIST **<pl>**, sa valeur associée prend la nouvelle valeur **<pval>**, sinon l'indicateur **<ind>** et sa valeur associée **<pval>** sont ajoutés ***en tête*** de **<pl>** (d'une manière identique à la fonction ADDPROP). PUT retourne **<pl>** en valeur.

PUT peut être défini en VLISP de la manière suivante :

```
(DE PUT (pl pval ind)
   (LET (p (PLIST pl))
      (COND
         ((NULL p) (ADDPROP pl pval ind))
         ((EQ (CAR p) ind) (RPLACA (CDR p) pval))
         (T (SELF (CDDR p)))))
   pl)
```

```
ex : (PLIST 'PLT)          ☞  (I1 A I2 B)
     (PUT 'PLT 'C 'I1)     ☞  PLT
     (PLIST 'PLT)          ☞  (I1 C I2 B)
     (PUT 'PLT 0 'I9)      ☞  PLT
     (PLIST 'PLT)          ☞  (I9 0 I1 C I2 B)
```

(REMPROP <pl> <ind>) [SUBR à 2 arguments]

enlève de la P-LIST **<pl>** l'indicateur **<ind>** s'il existe ainsi que sa valeur associée. REMPROP retourne **<pl>** en valeur.
L'utilisation conjuguée des fonctions REMPROP et ADDPROP permet d'utiliser les P-LIST comme des piles de propriété-valeurs.

REMPROP peut être défini en VLISP de la manière suivante :

```
(DE REMPROP (pl ind)
   (LET ((pl1 pl) (pl2 (PLIST pl)))
      (COND
         ((NULL pl2) ())
         ((EQ (CAR pl2) ind) (RPLACD pl1 (CDDR pl2)))
         (T (SELF pl2 (CDDR pl2))))))
```

```
ex : (PLIST 'PLT)          ☞  (I1 A I2 B)
     (REMPROP 'PLT 'I2)    ☞  PLT
     (PLIST 'PLT)          ☞  (I1 A)
```



3.12 LES FONCTIONS SUR LES P-NAME DES ATOMES

Rappelons que :
- le P-NAME d'un atome littéral est son nom externe
- le P-NAME d'un nombre est sa représentation externe dans la base de conversion de sortie courante.

Toutes ces fonctions peuvent être décrites au moyen des 2 fonctions de base, IMPLODE et EXPLODE qui permettent de passer de la représentation interne d'un P-NAME à sa représentation externe et vice-versa.

(EXPLODE <s>) [SUBR à 1 argument]

retourne la liste de tous les caractères de la représentation externe de l'expression **<s>**, qui peut être de type quelconque. EXPLODE retourne la liste de caractères qui serait imprimée si on demandait l'impression de **<s>** au moyen de la fonction PRIN (du reste il n'y a pas de fonction EXPLODE à proprement parler dans l'interprète mais un changement de direction du flux de sortie produit par la fonction PRIN).

```
ex : (EXPLODE '-237)          ☞  (- 2 3 7)
     (EXPLODE '(CAR '(A B)))
        ☞  (/( C A R /  /( Q U O T E /  /( A /  /B /) /) /) /))
```
dans cet exemple les caractères spéciaux parenthèse ouvrante, parenthèse fermante et espace sont quotés au moyen du caractère /

(IMPLODE <l>) [SUBR à 1 argument]

suppose que **<l>** est une liste de caractères (i.e. d'atomes mono-caractères). IMPLODE retourne l'objet VLISP qui possède comme représentation externe la liste de caractères **<l>**. IMPLODE est l'inverse de EXPLODE et permet de fabriquer de nouveaux objets VLISP au moyen de leur représentation externe DE LA MEME MANIERE que si ces caractères étaient lus par la fonction READ (il n'y a pas de fonction spéciale IMPLODE dans l'interprète mais simplement un changement d'origine du flux d'entrée de la fonction READ).

```
ex : (IMPLODE '(- 1 2 9))          ☞  -129
     (IMPLODE (EXPLODE '(A B)))  ☞  (A B)
```

(PLENGTH <at>) [SUBR à 1 argument] {pour Pname LENGTH}

retourne en valeur le nombre de caractères du P-NAME de l'atome littéral **<at>**. Si l'argument **<at>** n'est pas un atome littéral, PLENGTH retourne 0.

PLENGTH peut être défini en VLISP de la manière suivante :

```
(DE PLENGTH (at)
   (IF (LITATOM at)
      (LENGTH (EXPLODE at))
      0))
```

```
ex :  (PLENGTH 'FOOBAR)  ☞  6
      (PLENGTH)          ☞  3  taille de l'atome NIL
      (PLENGTH 120)      ☞  0
```

(CHRPOS <c> <at>) [SUBR à 2 arguments]

retourne en valeur la position du caractère **<c>** dans le P-NAME de l'atome littéral **<at>**. Si le caractère **<c>** n'existe pas dans le P-NAME de l'atome CHRPOS retourne NIL. La position (le rang) du premier caractère est 1 (et non 0).

CHRPOS peut être défini en VLISP de la manière suivante :

```
(DE CHRPOS (c at)
   (LET ((l (EXPLODE at)) (n 1))
      (COND
         ((NULL l) NIL)
         ((EQ (CAR l) c) n)
         (T (SELF (CDR l) (1+ n))))))
```

```
ex : (CHRPOS 'Y "OTYoty")                ☞ 3
     (CHRPOS 'N "OTY")                   ☞ NIL
     (CHRPOS 'D "0123456789ABCDEF")      ☞ 14
```

(CHRNTH <n> <at>) [SUBR à 2 arguments]

retourne le **<n>**ième caractère du P-NAME de l'atome littéral **<at>** et s'il n'existe pas retourne la valeur NIL.

CHRNTH peut être défini en VLISP de la manière suivante :

```
(DE CHRNTH (n at)
   (IF (LITATOM at)
      (CNTH n (EXPLODE at))
      ()))
```

```
ex : (CHRNTH 1 'FOO)                     ☞ F
     (CHRNTH 11 "0123456789ABCDEF")      ☞ A
```

(SORT <at1> <at2>) [SUBR à 2 arguments]

retourne T si le P-NAME de l'atome littéral **<a1>** est inférieur ou égal (lexicographiquement) au P-NAME de l'atome littéral **<a2>**, sinon retourne NIL. Cette fonction est utilisée pour réaliser des tris alphabétiques.

SORT peut être défini en VLISP de la manière suivante :

```
(DE SORT (at1 at2)
   (LET ((l1 (EXPLODE at1)) (l2 (EXPLODE at2)))
      (COND
         ((NULL l1) T)
         ((<= (CASCII (CAR l1)) (CASCII (CAR l2)))
            (SELF (CDR l1) (CDR l2)))
         (T ()))))
```

```
ex : (SORT 'A 'A)          ☞ T
     (SORT 'B 'A)          ☞ NIL
     (SORT 'A 'B)          ☞ T
     (SORT 'ZZZ 'ZZZZ)     ☞ T
```

```
; Comment réaliser le tri d'une liste par échange physique :

      (DE SORTL  (l ;; s)
         (SETQ s (APPEND l))
         (MAP (LAMBDA (sl1)
                (IF (CDR l)
                   (MAP (LAMBDA (sl2)
                          (IF (SORT (CAR sl1) (CAR sl2))
                             ()
                             (RPLACA sl2
                                (PROG1 (CAR sl1)
                                   (RPLACA sl1 (CAR sl2))))))
                        (CDR sl1))))
              S)
         S)

; Voici une autre manière de trier une liste d'atomes
; au moyen d'un tri-fusion récursif

   (DE TRI-FUSION (l)
      ; tri la liste l
      (IF (NULL (CDR l))
        l
        (FUSION (TRI-FUSION (MOITIE1 l)) (TRI-FUSION (MOITIE2 l)))))

   (DE FUSION (l1 l2)
      ; fusionne les 2 listes triées l1 et l2
      (COND
         ((NULL l1) l2)
         ((NULL l2) l1)
         ((SORT (CAR l1) (CAR l2))
             (CONS (CAR l1) (FUSION (CDR l1) l2)))
         (T  (CONS (CAR l2) (FUSION l1 (CDR l2))))))

   (DE MOITIE1 (l)
      ; retourne la 1ère moitié de l
      (LET ((l l) (n (// (LENGTH l) 2)))
         (IF (<= n 0)
            ()
            (CONS (CAR l) (SELF (CDR l) (1- n))))))

   (DE MOITIE2 (l)
      ; retourne la 2ème moitié de l
      (NTH (1+ (// (LENGTH l) 2)) l))

; Cette exemple n'est pas optimum quant au nombre de CONS effectués.
; L'utilisation des fonctions de modification physique permet la
; suppression de tous les CONS des fonctions FUSION et MOITIE1
```

(GENSYM <n>) [SUBR à 1 argument] *{pour GENerate SYMbol}*

retourne à chaque appel un nouvel atome littéral de type **Gxxx** dans lequel **xxx** est un nombre incrémenté de 1 à chaque appel (on appelle ce nombre le ***compteur*** de GENSYM); **xxx** vaut 100 au départ de l'interprète. Si l'argument **<n>** est fourni il devient le nouveau compteur de GENSYM.

si GENSYM-COUNT est le nom de la variable globale qui contient le compteur de GENSYM, il doit être initialisé :

```
(SETQ GENSYM-COUNT 100)
```

et GENSYM peut être défini en VLISP de la manière suivante :

```
(DE GENSYM (n)
   (IF n (SETQ GENSYM-COUNT n))
   (SETQ GENSYM-COUNT (1+ GENSYM-COUNT))
   (IMPLODE (CONS 'G (EXPLODE GENSYM-COUNT))))
```

```
ex : (GENSYM)          ☞ G101
     (GENSYM)          ☞ G102
     (GENSYM)          ☞ G103
```

Une utilisation intéressante de la fonction GENSYM est la possibilité de réaliser une protection contre les collisions homonymiques de certaines variables critiques contenues dans différents fichiers. Autrement dit, il faut pouvoir s'assurer que certaines variables d'un fichier seront ***uniques*** en ce sens qu'aucun autre atome ne pourra avoir le même P-NAME dans d'autres fichiers. Pour ce faire on peut définir un macro-caractère (ici le ~) qui sera placé devant toutes les occurrences de ces variables critiques. Pour réaliser cette protection de variables, durant la lecture d'un fichier, il faut l'organiser de la manière suivante :

```
; en tête du fichier faire

(SETQ prefix (EXPLODE (GENSYM)))

(DMC "~" () (IMPLODE (APPEND prefix (EXPLODE (READ)))))

     .....

(DE FOO (~l ~s )

     ; les variables l et s sont maintenant protégées
     ; car elles sont traduites : G101l  G101s ...
     .....

; et en fin de fichier provoquer l'irrémédiable

(GENSYM)
```

Enfin voici 2 fonctions de manipulation des codes internes des caractères.

(ASCII <n>) [SUBR à 1 argument]

retourne le caractère de code ASCII **<n>** (modulo 256).

```
ex : (ASCII 67)        ☞ C
     (1+ (ASCII 49))   ☞ 2
```

(CASCII <c>) [SUBR à 1 argument] *{pour Code ASCII}*

retourne le code ASCII du caractère **<c>**. Un caractère étant un atome dont le P-LEN est égal à 1.

```
ex : (CASCII 'C)   ☞ 67
     (CASCII 1)    ☞ 49
```

3.13 L'ARITHMETIQUE MIXTE

Certains systèmes (TRS80(TRSDOS) et TRS80(CP/M)) permettent d'utiliser des nombres flottants et tous les systèmes permettent d'utiliser des nombres entiers.
Le système VLISP 8 possède un ensemble de fonctions pouvant manipuler ces nombres. Les arguments de ces fonctions peuvent être des nombres de n'importe quel type : entiers ou flottants. Si l'un des arguments est flottant, le résultat de ces fonctions est un nombre flottant ; si tous les arguments sont de type entier, le résultat de ces fonctions est un nombre entier. Il y a donc conversion automatique des arguments avec priorité aux nombres flottants.

Ces fonctions ne testent pas le type des arguments. Toutefois tout argument absent (égal à NIL) est remplacé par la valeur 0.

Un débordement dans un calcul provoque l'erreur :

*** erreur débordement numérique. ou*
*** numeric overflow.*

Toutefois le test de débordement n'est validée que si la valeur de l'atome OVERFLOW est différente de NIL (ce qui est l'option par défaut). Si cet atome possède une valeur égale à NIL, aucun message n'apparait en cas de débordement et le calcul se poursuit avec des résultats bien évidemment faux.

Donc pour arrêter le programme en cas de débordement, il faut évaluer :

```
(SETQ OVERFLOW T)
```

sinon pour ne pas provoquer d'erreur, il faut évaluer :

```
(SETQ OVERFLOW)
```

Il est également possible de changer dynamiquement la valeur à l'atome OVERFLOW, par exemple :

```
(LET (OVERFLOW)
   <s1>          ; permet d'évaluer les différentes
   ....          ; formes <s1> ... <sN> sans craindre
   <sN>)         ; l'erreur ** débordement d'entier.
```

ATTENTION : d'une manière générale, les fonctions qui vont être décrites sont à préférer aux fonctions de la section suivante car elles existent dans tous les systèmes (que le système possède ou non des nombres flottants) et sont plus générales.

(1+ <n>) [SUBR à 1 argument]

retourne la valeur : <n> + 1 .

ex : (1+ 6) ☞ 7
(1+) ☞ 1

(1- <n>) [SUBR b 1 argument]

retourne la valeur : **<n>** - 1

```
ex : (1- 7)     ☞ 6
     (1-)       ☞ -1
```

(+ <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur de la somme : **<n1>** + **<n2>**

```
ex : (+ 5 6)       ☞ 11
     (+ 8)         ☞ 8
     (+)           ☞ 0
```

Certains VLISP possèdent la fonction PLUS à N arguments qui réalise une addition multiple. Cette fonction peut être définie en VLISP 8 avec la MACRO suivante :

```
(DM PLUS (l)
   (RPLACB l
      (LIST '+ (CADR l)
         (IF (NULL (CDDDR l))
            (CADDR l)
            (CONS 'PLUS (CDDR l))))))
```

ex : l'appel (PLUS z 12 y) est traduit (+ z (+ 12 y))

(- <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur de la différence : **<n1>** - **<n2>**

```
ex : (- 6 2)       ☞ 4
     (- 12)        ☞ 12
     (-)           ☞ 0
```

(* <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du produit : **<n1>** * **<n2>**

```
ex : (* 10 4)     ☞ 40
     (* 100)      ☞ 0
```

Certains VLISP possèdent la fonction TIMES à N arguments qui réalise une multiplication multiple. Cette fonction peut être définie en VLISP 8 avec la MACRO suivante :

```
(DM TIMES (l)
   (RPLACB l
      (LIST '* (CADR l)
         (IF (NULL (CDDDR l))
            (CADDR l)
            (CONS 'TIMES (CDDR l))))))
```

ex : l'appel (TIMES W X Y Z) est traduit (* W (* X (* Y Z)))

(/ <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du quotient de : **<n1> / <n2>.** Si l'argument **<n2>** est égal à 0, une erreur se produit dont le libellé est :

***** erreur division par 0.   ou***
***** 0 divide.***

Dans certaines versions de VLISP 8, la caractère / est un caractère spécial qui permet de quoter des caractères en entrée (voir les fonctions d'entrée). Dans le cas il faut soit quoter ce caractère lui-même au moyen d'un double slash //, soit utiliser la pseudo-chaîne "/".

```
ex : (/ 20 5)        ☞  4
     (/ 10)          ☞  ** erreur division par 0
     et si / est un caractère spécial
     (// 100 5)      ☞  20      ou bien
     ("/" 100 5)     ☞  20
```

(\ <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du reste de la division de **<n1>** par **<n2>.** Sur les claviers des systèmes TRS80, le caractère \ est obtenue par la frappe du caractère flèche en bas.

```
ex : (\ 11 3)     ☞  2
     (\ 12 4)     ☞  0
```

Il est souvent utile d'avoir des fonctions incrémentation et décrémentation. Celles-ci sont aisément réalisées sous forme de MACRO dont voici les textes :

```
(DM INCR (l)
   (RPLACB l
      (LIST 'SETQ (CADR l)
         (IF (NULL (CADDR l))
            (LIST '1+ (CADR l))
            (LIST '+ (CADR l) (CADDR l))))))

(DM DECR (l)
   (RPLACB l
      (LIST 'SETQ (CADR l)
         (IF (NULL (CADDR l))
            (LIST '1- (CADR l))
            (LIST '- (CADR l) (CADDR l))))))
```

exemple d'appel :

```
(INCR x 2)    est traduit   (SETQ x (+ x 2))
(DECR cpt)    est traduit   (SETQ cpt (1- cpt))
```

3.14 L'ARITHMETIQUE ENTIERE

Les fonctions qui vont être décrites utilisent des opérandes **<n>** supposés de type entier. Ces fonctions n'effectuent aucun contrôle de validité de type. Si les arguments ne sont pas des nombres entiers, ces fonctions livrent en général de bien étranges résultats.

L'utilisation de ces fonctions n'est pas recommandée pour des travaux ordinaires. En effet elles ne sont utilisées que par des programmes systèmes (comme le compilateur) pour des raisons d'efficacité.

Tout comme les fonctions du chapitre précédent, un débordement dans les calculs provoque l'erreur :

*** erreur débordement numérique.   ou***
*** numeric overflow.***

Et comme les fonctions précédentes, l'atome OVERFLOW contrôle l'impression d'un message et l'arrêt de l'évaluation en cas de débordement.

(ABS <n>) [SUBR à 1 argument]

retourne la valeur absolue de l'argument **<n>**.

ex : (ABS 10) ☞ 10
(ABS -10) ☞ 10
(ABS ()) ☞ 0

(MINUS <n>) [SUBR à 1 argument]

retourne la valeur : - **<n>**

ex : (MINUS -10) ☞ 10
(MINUS 25) ☞ -25
(MINUS ()) ☞ 0

(ADD1 <n>) [SUBR à 1 argument]

retourne la valeur : **<n>** + 1 .

ex : (ADD1 6) ☞ 7
(ADD1 -4) ☞ -3

(SUB1 <n>) [SUBR à 1 argument]

retourne la valeur : **<n>** - 1

ex : (SUB1 7) ☞ 6
(SUB1 -9) ☞ -10

(ADD <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur de la somme : **<n1> + <n2>**

```
ex : (ADD 5 6)      ☞  11
     (ADD -5 -6)    ☞  -11
     (ADD 8)        ☞  8
```

(SUB <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur de la différence : **<n1> - <n2>**

```
ex : (SUB 6 2)      ☞  4
     (SUB 12 -7)    ☞  19
     (SUB 12)       ☞  12
```

(MUL <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du produit : **<n1> * <n2>**

```
ex : (MUL 10 4)        ☞  40
     (MUL 2 -3)        ☞  -6
     (MUL -100 -100)   ☞  10000
```

(DIV <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du quotient de : **<n1> / <n2>.**
DIV effectue une division entière sur des nombres entiers. Si l'argument **<n2>** est égal à 0, une erreur se produit dont le libellé est :

***** erreur division par 0.*** ***ou***
***** 0 divide.***

```
ex : (DIV 23 5)     ☞  4
     (DIV 40 -4)    ☞  -10
     (DIV 10)       ☞  ** erreur division par 0
```

(REM <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du reste de la division entière de **<n1>** par **<n2>.**

```
ex : (REM 11 3)   ☞  2
     (REM 14 22)  ☞  14
     (REM 12 0)   ☞  ** erreur division par 0
```

3.15 LES TESTS DE L'ARITHMETIQUE MIXTE

Ces fonctions possèdent toutes 2 arguments **<n1>** et **<n2>** qui peuvent être des nombres de n'importe quel type (entier ou flottant). Si l'un des arguments est flottant, ces fonctions réalisent des comparaisons flottantes ; si tous les arguments sont de type entier, les fonctions réalisent des comparaisons entières. Il y a donc conversion automatique des arguments avec priorité aux nombres flottants.

Si un argument n'est pas fourni à l'appel (si l'argument = NIL) il est remplacé par la valeur 0.

ATTENTION : d'une manière générale, les fonctions qui vont être décrites sont à préférer aux fonctions de la section suivante car elles existent dans tous les systèmes (que le système possède ou non des nombres flottants) et sont plus générales.

Toutes ces fonctions retournent le 1er argument si le test est vérifié et NIL dans le cas contraire.

(= <n1> <n2>) [SUBR à 2 arguments] *ou bien*

si **<n1>** = **<n2>** alors = retourne **<n1>**, sinon = retourne NIL. La fonction ZEROP n'a pas été conservée dans VLISP 8 car elle est équivalente à la forme (= **<n>**).

```
ex : (= 10 10)      ☞ 10
     (= -3)         ☞ NIL
     (= x) est équivalent à l'ancienne forme VLISP (ZEROP x)
```

(<> <n1> <n2>) [SUBR à 2 arguments] *ou bien*

si **<n1>** ≠ **<n2>** alors <> retourne **<n1>**, sinon <> retourne NIL.

```
ex : (<> 11 10)     ☞ 11
     (<> -3 -3)     ☞ NIL
```

(>= <n1> <n2>) [SUBR à 2 arguments]

si **<n1>** >= **<n2>** alors >= retourne **<n1>** sinon >= retourne NIL.

```
ex : (>= 3 7)       ☞ NIL
     (>= 7 7)       ☞ 7
```

(> <n1> <n2>) [SUBR à 2 arguments]

si **<n1>** > **<n2>** alors > retourne **<n1>** sinon > retourne NIL.

```
ex : (> 5 5)        ☞ NIL
     (> 7 4)        ☞ 7
```

(<= <n1> <n2>) [SUBR à 2 arguments]

si **<n1> <= <n2>** alors <= retourne **<n1>** sinon <= retourne NIL.

```
ex : (<= 5 5)     ☞ 5
     (<= 4 6)     ☞ 4
     (<= 8)       ☞ NIL
```

(< <n1> <n2>) [SUBR à 2 arguments]

si **<n1>** < **<n2>** alors < retourne **<n1>** sinon < retourne NIL. La fonction MINUSP n'a pas été conservée dans VLISP 8 car elle est équivalente à la forme (< **<n>**).

```
ex : (< 5 5)     ☞ NIL
     (< 4 5)     ☞ 4
     (< 0)       ☞ NIL
```

3.16 LES TESTS DE L'ARITHMETIQUE ENTIERE

Ces fonctions possèdent toutes 2 arguments **<n1>** et **<n2>** qui doivent être des nombres de type entiers. Elles n'effectuent aucun test de validité de type et ne provoquent donc jamais d'erreur. Toutefois si les arguments de ces fonctions ne sont pas des nombres entiers, leurs résultats ne sont pas significatifs.
Si un argument n'est pas fourni à l'appel (si l'argument = NIL) il est remplacé par la valeur entière 0.

Toutes ces fonctions retournent le 1er argument si le test est vérifié et NIL dans le cas contraire.

L'utilisation de ces fonctions n'est pas recommandée pour des travaux ordinaires. En effet elles ne sont utilisées que par des programmes systèmes (comme le compilateur) pour des raisons d'efficience.

(EQN <n1> <n2>) [SUBR à 2 arguments] *{pour EQ Number}*

si **<n1>** = **<n2>** alors EQN retourne **<n1>**, sinon EQN retourne NIL.

```
ex : (EQN 10 10)    ☞ 10
     (EQN -3)       ☞ NIL
```

(NEQN <n1> <n2>) [SUBR à 2 arguments] *{pour NEQ Number}*

si **<n1>** ≠ **<n2>** alors NEQN retourne **<n1>**, sinon NEQN retourne NIL.

```
ex : (NEQN 11 10)    ☞ 11
     (NEQN -3 -3)    ☞ NIL
```

(GE <n1> <n2>) [SUBR à 2 arguments]

si **<n1>** >= **<n2>** alors GE retourne **<n1>** sinon GE retourne NIL.

```
ex : (GE 3 7)      ☞  NIL
     (GE 7 7)      ☞  7
```

(GT <n1> <n2>) [SUBR à 2 arguments]

si **<n1>** > **<n2>** alors GT retourne **<n1>** sinon GT retourne NIL.

```
ex : (GT 5 5)     ☞  NIL
     (GT 7 4)     ☞  7
```

(LE <n1> <n2>) [SUBR à 2 arguments]

si **<n1>** <= **<n2>** alors LE retourne **<n1>** sinon LE retourne NIL.

```
ex : (LE 5 5)      ☞  5
     (LE 4 6)      ☞  4
     (LE 8)        ☞  NIL
```

(LT <n1> <n2>) [SUBR à 2 arguments]

si **<n1>** < **<n2>** alors LT retourne **<n1>** sinon LT retourne NIL.

```
ex : (LT 5 5)     ☞  NIL
     (LT 4 5)     ☞  4
     (LT 0)       ☞  NIL
```



3.17 LES FONCTIONS LOGIQUES

Pour toutes les fonctions qui vont être décrites, les arguments **<n1>** et **<n2>** doivent être de type entier. Ces fonctions ne travaillent que sur des OCTETS (donc que sur les 8 bits de poids faibles des nombres), ne réalisent aucune conversion de type et ne provoquent jamais d'erreur.

(COMPL <n>) [SUBR à 1 argument]

retourne la valeur du complément logique de **<n>**.

(COMPL <n>) est équivalent à (LOGXOR <n> #FF).

ex : (COMPL 0) ☞ 255 en héxadécimal #00FF
(COMPL -2) ☞ 1

(LOGAND <n1> <n2>) [SUBR à 2 arguments] *{pour LOGical AND}*

effectue l'opération de ET logique entre **<n1>** et **<n2>**.

ex : (LOGAND #36 #25) ☞ #24

pour savoir si <n> est une puissance de 2 évaluez :

(= <n> (LOGAND <n> (MINUS <n>)))

(LOGOR <n1> <n2>) [SUBR à 2 arguments] *{pour LOGical OR}*

effectue l'opération de OU logique entre **<n1>** et **<n2>**.

ex : (LOGOR #15 #7) ☞ #17

(LOGXOR <n1> <n2>) [SUBR à 2 arguments] *{pour LOGical XOR}*

effectue l'opération de OU exclusif logique entre **<n1>** et **<n2>**.

ex : (LOGXOR 5 3) ☞ 6

Il n'y a pas de décalages logiques en VLISP 8, ces fonctions doivent être écrites sous forme d'EXPR.

```
; ATTENTION : il faut empêcher les erreurs de débordement
; d'entiers AVANT d'utiliser ces décalages.

(DE LOGSHIFT (n nb ;; OVERFLOW)
   ; décale logiquement la valeur n de nb positions.
   ; Si nb est >0, le décalage s'effectue à gauche,
   ; sinon si nb <0, le décalage s'effectue à droite.
   (IF (= nb 0)
       n
       (IF (< nb 0)
           (LOGSHIFT (// n 2) (1+ nb))
           (LOGSHIFT (* n 2) (1- nb)))))
```

IV - LES ENTREES/SORTIES

Les entrées/sorties s'effectuent soit au niveau du caractère soit au niveau des S-expressions VLISP, sur des flux séquentiels.

Toutefois de nombreuses fonctions ont été rajoutées pour pouvoir utiliser des périphériques spéciaux tels :
- des terminaux graphiques
- des boîtes à musique
- des télévisions en couleur

4.1 LES FONCTIONS D'ENTREE DE BASE

Toutes les lectures sont réalisées dans le flux d'entrée qui est associé soit au terminal, soit à un fichier sélecté par la fonction INPUT, soit à une bande magnétique (de type cassette).

Voici les fonctions d'entrée de base.

(READ) [SUBR à 0 argument]

lit la S-expression suivante de type quelconque (atome ou liste) du flux d'entrée et la retourne en valeur. READ est la principale fonction de lecture et permet de lire des objets VLISP. Son fonctionnement détaillé est donné dans la section suivante.
Si une erreur de syntaxe VLISP est détectée, une erreur fatale se produit et le libellé suivant est imprimé :

***** erreur de syntaxe.   ou***
***** syntax error.***

Le plus souvent cette erreur provient d'une mauvaise utilisation des paires pointées ou d'un P-NAME trop long (de plus de 62 caractères). du a l'oubli du caractère ".

(READLINE) [SUBR à 0 argument]

lit la ligne suivante du flux d'entrée et retourne la liste des caractères de cette ligne. Cette liste ne contient pas le caractère fin de ligne (en l'occurrence le caractère Return) ni les caractères line-feed. READLINE est utilisée pour réaliser à peu de frais les interfaces entre programme et utilisateur.

ATENTION : cette fonction n'effectue pas de conversion automatique des caractères minuscules en caractères majuscules (pour réaliser ette conversion automatiquement, positionnez le bit 4 du STATUS READ).

READLINE peut être défini en VLISP de la manière suivante :

```
(DE READLINE ()
   (FREVERSE
      (LET ((l)) (c (READCH)))
         (COND
            ((EQ (CASCII c) #0D)
               ; code du Return
               l)
            ((EQ (CASCII c) #0A)
               ; code du Line/Feed
               (SELF l (READCH)))
            (T (SELF (CONS c l) (READCH)))))))
```

```
ex :  ? (READLINE)                ; appelle la lecture d'une ligne
      Le gai rossignol            ; ligne terminée par Return
      = (L e /  g a i /  r o s s i g n o l ) ; liste des caractères lus

      ; pour fabriquer une suite de mots à partir d'une ligne
      ; qui ne contient pas de caractères spéciaux ( ) . [ ]

      ? (IMPLODE (APPEND "(" (APPEND (READLINE) ")")))
      Le merle moqueur            ; ligne terminée par Return
      = (Le merle moqueur)        ; liste des atomes lus
```

(READCH) [SUBR à 0 argument] *{pour READ CHaracter}*

lit et retourne en valeur le caractère suivant du flux d'entrée. Ce caractère est retourné sous la forme d'un atome (littéral pour les lettres et numérique pour les chiffres).
Cette fonction n'effectue pas de conversion automatique des caractères minuscules en caractères majuscules.

(PEEKCH) [SUBR à 0 argument] *{pour PEEK CHaracter}*

retourne en valeur le caractère du flux d'entrée d'une manière identique à la fonction READCH, toutefois, ce caractère n'est pas véritablement lu mais seulement "consulté". Il en résulte que des appels successifs de la fonctions PEEKCH retournent toujours le même résultat. PEEKCH est utilisée pour "sentir" le caractère suivant du flux d'entrée AVANT de le lire véritablement.

(TEREAD) [SUBR à 0 argument] *{pour TErminate READ}*

passe à l'enregistrement suivant du flux d'entrée (une ligne sur TTY, une carte perforée, ...) en ignorant le reste éventuel de l'enregistrement courant. Cette fonction permet d'effacer le contenu du tampon d'entrée. TEREAD retourne NIL en valeur.

4.2 CONTROLE DES FONCTIONS D'ENTREE

4.2.1 Utilisation du terminal en entrée

A chaque demande de ligne sur le terminal, le système imprime le caractère ? puis un certain nombre d'espaces en fonction de la profondeur de la lecture (i.e. du nombre de parenthèses ouvrantes non encore refermées). Ce dispositif connu sous le nom de PRETTY-READ (qui est propre à VLISP) permet de contrôler immédiatemment le niveau d'imbrication des parenthèses.

De plus un éditeur de ligne est activé. Il en existe de 2 types en fonction du système utilisé :
- l'éditeur de ligne de type TRS80
- l'éditeur de ligne standard.

L'éditeur de ligne de type TRS80 interprète les caractères suivants :
- ← : détruit le dernier caractère entré
- SHIFT/← : détruit toute la ligne entrée
- BREAK : provoque l'appel de la fonction BREAK qui d'une manière standard retourne au top-level de l'interprète.
- ENTER : termine la ligne qui est envoyée aux fonctions de lecture

L'éditeur de ligne standard interprète les caractères suivants :
- RUBOUT ou DELETE ou ← (Backspace) : détruit le dernier caractère entré.
- ↑U ou ↑X : détruit toute la ligne entrée
- ↑R : reimprime le contenu de la ligne courante (cette commande est utile sur les terminaux papier pour s'assurer du contenu du tampon après les corrections).
- ↑C : provoque l'appel de la fonction BREAK qui d'une manière standard retourne au top-level de l'interprète.
- RETURN : termine la ligne qui est envoyée aux fonctions de lecture.

Il existe dans les systèmes TRS 80, un véritable éditeur sur écran inclu dans l'interprète lui-même. Sa description est donnée au chapitre 6.

4.2.2 La lecture standard

La lecture des S-expressions VLISP par la fonction READ s'effectue en format LIBRE i.e. que chaque élément syntaxique peut être encadré d'un ou plusieurs espaces.

Durant la lecture des atomes littéraux seuls les 62 premiers caractères sont pris en compte. Il y a transcodage automatique des caractères minuscules en caractères majuscules. Les caractères de contrôle (i.e. les caractères entrés en pressant simultanément la touche CNTRL et le caractère) sont imprimés *à la DEC* (i.e. ↑ suivi du caractère). S'il faut insérer des caractères spéciaux dans un atome littéral, il faut les faire précéder du caractère spécial quote-caractère / (le slash).
Une autre solution consiste à utiliser les ***pseudo-chaînes de caractères***. Une pseudo-chaîne de caractères est représentée par un suite de caractères quelconques encadrée du caractère délimiteur de chaîne (par défaut le "). N'importe quel caractère peut faire partie d'une chaîne en particulier les caractères de mouvement de papier, Return, Line-feed ... Il n'y a jamais de transcodage minuscule majuscule durant la lecture des chaînes.
Ces pseudo-chaînes sont considérées comme des ATOMES LITTERAUX normaux, mais possèdent une propriété remarquable : à leur création elles sont considérées comme des constantes (et possèdent leur propre valeur en C-VAL). Ce dernier trait permet d'éviter de "quoter" ces atomes et d'être

ainsi compatible avec les interprètes qui possèdent réellement des chaînes de caractères (en particulier VLISP 10).

ex : CaR	correspond à l'atome	CAR
LONGTRESLONGATOME	" "	LONGTRESLONGATOME
RE/(3/)	" "	RE(3)
"Tres bizar. ?/#()"		Tres bizar. ?/#()

Les nombres entiers sont représentés par une suite de chiffres décimaux qui peut être précédée du signe + ou du signe -. Les nombres héxadécimaux sont représentés par une suite de chiffres hexadécimaux (chiffre décimal de 0 ... 9 ou lettre de A ... F) précédée du caractère spécial # (le dièze).

ex : 123	correspond au nombre	123
+27	" "	27
-45	" "	-45
#12	" "	18
#FF	" "	255
+	n'est pas un nombre	

Il est possible d'insérer des commentaires, qui sont totalement ignorés au cours de la lecture. Un commentaire est une suite de caractères quelconques précédée d'un caractère de type début de commentaire et terminée par le caractère de type fin de commentaires OU de nouveau par un caractère de type début de commentaires. Par défaut il n'y a qu'un caractère de type début de commentaires, le caractère point-virgule (;) et qu'un seul caracrère fin de commentaires le caractère Return. Par défaut tout commentaire s'arrêtera donc en fin de ligne.

```
                                                    commentaires :
ex : (DE FOO (N ;le nombre; L ;la liste;)         - au milieu de la ligne
        (IF (= N 0)      ; yen a pu à faire       - en fin de ligne
           ...
```

La représentation des listes est classique : une liste se représente par une parenthèse ouvrante "(" suivie des éléments de la liste séparés par des espaces et suivis d'une parenthèse fermante ")". Il est possible également d'utiliser la notation pointée généralisée :

(S-expr . S-expr)

```
ex :  (A . (B . (C . D))) correspond à (A B C . D)
      ((A) . (B))              "     "   ((A) B)
```

Au début de la fonction READ, il peut y avoir un nombre quelconque de parenthèses fermantes qui sont ignorées. Ceci permet de refermer à coup sûr les S-expressions avec une giclée de parenthèses fermantes sans avoir à les dénombrer.

(DE FOO (N) (ADD1 N)))))) est lu sans erreur

Une fonction STATUS permet de régler certaines modalités de la lecture.

(STATUS READ <n>) [FSUBR]

permet de connaître (si la valeur **<n>** n'est pas fournie) ou de positionner (si la valeur **<n>** est fournie) les indicateurs internes de la fonction READ. Chaque indicateur est représenté par un bit dans la valeur **<n>**. A l'initialisation du système, tous ces indicateurs valent 0 (on a donc implicitement l'appel (STATUS READ 0)).

Signification des bits du STATUS READ :

- bit 0 = 0 si il faut faire l'écho de tous les caractères lus; bit 0 = 1 s'il n'y a pas d'écho à effectuer.

- bit 1 = 0 si le périphérique d'entrée est un terminal (TTY); bit 1 = 1 si le périphérique d'entrée est un lecteur de ruban perforé.

- bit 2 = 0 si le périphérique d'entrée est un périphérique à écran (CRT); bit 2 = 1 si le périphérique n'est pas un écran.

- bit 3 = 0 si le périphérique d'entrée à écran ne fait pas l'écho des caractères DELete ou RUBOUT; bit 3 = 1 si le terminal effectue l'écho.

  ex : réalisation d'une fonction de lecture sans écho :

```
(DE LIRAVEUGLE ()
   (STATUS READ 1)
   (PROG1 (READ) (STATUS READ 0)))
```

- bit 4 = 0 si le périphérique possède des minuscules. Si ce bit est à 1 tous les caractères minuscules sont dès leur lecture sur le terminal convertit en caractères majuscules.

  ex : fonction de lecture d'une ligne en capitale

```
(DE LIRCAP ( ;; vieuxstatus)
   (SETQ vieuxstatus (STATUS READ))
   (STATUS READ (LOGOR vieuxstatus #10))
   (PROG1 (READLINE) (STATUS READ vieuxstatus)))
```

4.2.3 Les macro-caractères

Un macro-caractère est un caractère quelconque auquel a été associé une fonction qui est lancée automatiquement ***à la lecture de ce caractère dans le flux d'entrée.*** La valeur retournée par cette fonction remplace le macro-caractère lu. Tout caractère peut être utilisé comme macro-caractère.

Il existe 3 macro-caractères standards :

- le quote (apostrophe) ' qui, placé devant une S-expression quelconque, retourne la liste (QUOTE S-expression).
  ex : '(A B) est équivalent à (QUOTE (A B))
  ''A " (QUOTE (QUOTE A))

- le contrôle-C ↑C qui provoque l'abandon de la lecture en invoquant la fonction système BREAK.

- le & (et commercial) qui permet d'appeler directement l'éditeur vidéo des systèmes TRS 80.

Le caractère de spécification de nombre en hexadécimal (le #) n'est pas un macro-caractère mais bien un caractère spécial.

Il existe une fonction pour définir un macro-caractère.

(DMC <c> <l> <s1> ... <sN>) [FSUBR]
{pour Define Macro Character}

l'argument **<c>** doit être un atome littéral mono-caractère. DMC associe à ce caractère une fonction qui possède une liste de variables locales **<l>** (cette liste est obligatoire à cette position même s'il n'y a pas de variables locales) et un corps de fonction **<s1> ... <sN>**. DMC possède la même syntaxe que les autres fonctions de définition (DE, DF et DM) et retourne **<c>** en valeur.

ex : si les macro-caractères standards n'étaient pas définis, on pourrait le faire de la manière suivante :

```
(DMC "'" ()
   (LIST QUOTE (READ)))      ; retourne la liste (QUOTE <S-expr lue>) ;

(DMC "↑C" ()
   (BREAK))

(DMC "&" ()
   (LIST 'EDITV (READ)))     ; retourne la liste (EDITV <S-expr lue>) ;
```

La fonction DMC peut être défini en VLISP :

```
(DF DMC (L)
   (EVAL (CONS 'DE L))       ; fabrique la fonction associée
   (TYPECH (CAR L) 5))       ; force le nouveau type de ce caractère
   (CAR L))                  ; et retourne en valeur le caractère.
```

Pour détruire une définition de macro-caractère, il faut changer le type du caractère et détruire la fonction qui lui était associée par exemple au moyen de la fonction suivante :

```
(DE REMACH (c)               ; {pour REMove MAcro CHaracter}
   (TYPECH c 8)              ; le caractère redevient normal
   (FTYPE c 0))              ; détruit l'ancienne définition.
```

ATTENTION : la fonction associée à un macro-caractère étant de la même nature que les fonctions définies par l'utilisateur (de type DE) il n'est pas possible, pour un atome mono-caractère, de posséder tout à la fois une définition de type DE et une définition de type DMC.

4.2.4 Type des caractères

L'analyseur lexical VLISP (i.e. la fonction READ) utilise une "table de lecture" pour effectuer commodément son analyse. Cette table associe un type codé à chacun des caractères.
Cette table de lecture est totalement accessible à l'utilisateur qui peut ainsi la changer pour pouvoir lire facilement de nouveaux dialectes de LISP aux syntaxes étranges.

Les types de caractères disponibles sont les suivants :

0 : type NULL. Tous les caractères de ce type sont complètement ignorés à la lecture (Ex: le caractère Line/Feed, le caractère "Avance-bande" ou NULL ...).

1 : type QUOTEC. Ce type de caractère est utilisé pour "quoter" n'importe quel autre caractère. "Quoter" un caractère consiste à lui donner implicitement le type 8 (le type des caractères normaux). Par défaut il n'existe qu'un seul caractère de ce type, le caractère "slash" / .

2 : type BCOM. Ce type de caractère sert à indiquer le début d'un commentaire qui sera terminé à l'occurrence d'un caractère de ce même type ou d'un caractère du type suivant. Par défaut il n'existe qu'un seul caractère de ce type, le caractère point virgule ; .

3 : type ECOM. Ce type de caractère sert à indiquer la fin d'un commentaire. Par défaut il n'existe qu'un seul caractère de ce type, le caractère RETURN

4 : type SEP. Définit un caractère séparateur standard (Ex: l'espace, la tabulation ...).

5 : type MACH. Ce type indique que le caractère est un macro-caractère. Une fonction est associée à l'atome dont le P-NAME est ce caractère. Cette fonction est invoquée automatiquement à la lecture de ce caractère dans le flux d'entrée.

6 : type CSTRING. Ce type de caractère fait office de caractère délimiteur de pseudo-chaîne. Par défaut il n'existe qu'un seul caractère de ce type, le caractère guillemets " .

7 : type NHEX. Ce type de caractère sert de spécificateur de nombre représenté en hexadécimal. Par défaut ce caractère est le dièze # .

8 : type NORMAL. Définit un caractère normal pouvant être utilisé pour construire un P-NAME.

9 : type DOT. Ce type de caractère (le point . par défaut) sert à écrire les paires pointées.

10 : type LPAR. Ce type de caractère (la parenthèse ouvrante (par défaut) sert de caractère de début de liste.

11 : type RPAR. Ce type de caractère (la parenthèse fermante) par défaut) sert de caractère de fin de liste.

12 : type LBRA. Ce type de caractère (le crochet ouvrant [par défaut) indique le début d'une liste évaluée.

13 : type RBRA. Ce type de caractère (le crochet fermant] par défaut) indique la fin d'une liste évaluée.

(TYPECH <c> <n>) [SUBR à 2 arguments]

permet de connaître (si **<n>** n'est pas fourni) ou de modifier (si **<n>** est fourni) le type du caractère **<c>**. Cette fonction retourne le type courant du caractère **<c>** après modification éventuelle.

ex : ; après la redéfinition syntaxique des
; caractères < et >,

```
(TYPECH '< 10)   ☞ 10
(TYPECH '> 11)   ☞ 11
```

; l'entrée :

```
<CONS '(A> '<B)>
```

; est lue :

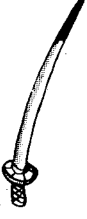
```
(CONS '(A) '(B))
```

Voici la table des caractères standards

#0	0		0	
#1	1	↑A	8	
#2	2	↑B	8	
#3	3	↑C	5	(LAMBDA NIL (BREAK))
#4	4	↑D	8	
#5	5	↑E	8	
#6	6	↑F	8	
#7	7	bell	8	
#8	8	bs	1	
#9	9	tab	4	
#A	10	lf	4	
#B	11	vt	4	
#C	12	ff	4	
#D	13	cr	3	
#E	14	↑N	8	
#F	15	↑O	8	
#10	16	↑P	8	
#11	17	↑Q	8	
#12	18	↑R	8	
#13	19	↑S	8	
#14	20	↑T	8	
#15	21	↑U	8	
#16	22	↑V	8	
#17	23	↑W	8	
#18	24	↑X	8	
#19	25	↑Y	8	
#1A	26	↑Z	8	
#1B	27	~	8	
#1C	28	V	8	
#1D	29	L	8	
#1E	30	S	2	
#1F	31	P	8	
#20	32		4	
#21	33	!	8	
#22	34	"	6	

#23	35	#	7	
#24	36	$	8	
#25	37	%	8	
#26	38	&	5	(LAMBDA NIL (LIST (QUOTE EDITV) (READ)))
#27	39	'	5	(LAMBDA NIL (LIST QUOTE (READ)))
#28	40	(	10	
#29	41	)	11	
#2A	42	*	8	
#2B	43	+	8	
#2C	44	,	8	
#2D	45	-	8	
#2E	46	.	9	
#2F	47	/	1	
#30	48	0	8	
#31	49	1	8	
#32	50	2	8	
#33	51	3	8	
#34	52	4	8	
#35	53	5	8	
#36	54	6	8	
#37	55	7	8	
#38	56	8	8	
#39	57	9	8	
#3A	58	:	8	
#3B	59	;	2	
#3C	60	<	8	
#3D	61	=	8	
#3E	62	>	8	
#3F	63	?	8	
#40	64	@	8	
#41	65	A	8	
#42	66	B	8	
#43	67	C	8	
#44	68	D	8	
#45	69	E	8	
#46	70	F	8	
#47	71	G	8	
#48	72	H	8	
#49	73	I	8	
#4A	74	J	8	
#4B	75	K	8	
#4C	76	L	8	
#4D	77	M	8	
#4E	78	N	8	
#4F	79	O	8	
#50	80	P	8	
#51	81	Q	8	
#52	82	R	8	
#53	83	S	8	
#54	84	T	8	
#55	85	U	8	
#56	86	V	8	
#57	87	W	8	
#58	88	X	8	
#59	89	Y	8	
#5A	90	Z	8	
#5B	91	[	12	
#5C	92	\	8	
#5D	93	]	13	
#5E	94	↑	8	
#5F	95	←	8	
#60	96	ç	8	
#61	97	a	8	

#62	98	b	8
#63	99	c	8
#64	100	d	8
#65	101	e	8
#66	102	f	8
#67	103	g	8
#68	104	h	8
#69	105	i	8
#6A	106	j	8
#6B	107	k	8
#6C	108	l	8
#6D	109	m	8
#6E	110	n	8
#6F	111	o	8
#70	112	p	8
#71	113	q	8
#71	114	r	8
#73	115	s	8
#74	116	t	8
#75	117	u	8
#76	118	v	8
#77	119	w	8
#78	120	x	8
#79	121	y	8
#7A	122	z	8
#7B	123	{	8
#7C	124	|	8
#7D	125	}	8
#7E	126	~	8
#7F	127		0





4.3 LES FONCTIONS DE SORTIE DE BASE

Toutes éditent dans un tampon de sortie totalement accessible à l'utilisateur qui est vidé dans le flux de sortie associé soit à un terminal soit à un fichier sélecté au moyen de la fonction OUTPUT.
Si au cours d'une édition le tampon de sortie devient plein, il est automatiquement imprimé, en utilisant la fonction (TERPRI 1), et l'édition se poursuit dans un nouveau tampon vide.

On peut, à tout moment, suspendre une impression sur le terminal (holding) en entrant sur le terminal un caractère spécial, faire repartir l'impression en entrant un autre caractère ou bien appeler la fonction système BREAK durant une impression.

Les caractères utilisés dépendent du système :

- avec les systèmes TRS 80 le caractère SHIFT/@ suspend l'impression, le caractère SHIFT/@ (le même donc) la fait reprendre et la touche BREAK appelle la fonction BREAK.

- avec les autres systèmes, le caractère ↑S suspend l'impression, le caractère ↑Q la fait reprendre et le caractère ↑C appelle la fonction BREAK.

(PRIN <s1> ... <sN>) [FSUBR à N arguments]

édite dans le tampon de sortie la valeur de l'évaluation des différentes S-expressions **<s1> ... <sN>** sans imprimer le tampon. PRIN retourne en valeur la valeur de l'évaluation de **<sN>**.

(TERPRI <n>) [SUBR à 1 argument] *{pour TERminate PRInt}*

imprime tous les caractères du tampon de sortie, se positionne en début de ligne en imprimant le code Return , saute **<n>** lignes en imprimant **<n>** fois le code Line-feed , puis édite un certain nombre d'espaces pour se positionner à la marge gauche (voir la fonction LMARGIN). Si l'argument **<n>** n'est pas un nombre ou n'est pas fourni, TERPRI agit comme si **<n>** était égal à 1 provoquant une impression avec interlignage simple. L'appel (TERPRI) est donc équivalent à (TERPRI 1).
Si **<n>** = 0, le positionnement en début de ligne (au moyen du caractère return) est quand même réalisé mais aucune ligne n'est sautée et il n'y pas de positionnement à la marge gauche ce qui permet de réaliser une surimpression . Attention toutefois aux tentatives de surimpression sur des terminaux à écran...
Si **<n>** = -1, le positionnement en début de ligne et à la marge gauche n'a pas lieu et aucune ligne n'est sautée. Cette option permet d'imprimer des messages et d'attendre des réponses sur la même ligne ce qui facilite les dialogues.

```
ex : ? (DE QUAM (msg)
     ?    (PRIN msg)        ; édite le msg dans le tampon
     ?    (TERPRI -1)       ; puis imprime le contenu du tampon.
     ?    (READ))           ; enfin lecture de la réponse.
     = QUAM

     ? (QUAM "Nb de satellites de VEGA")
       Nb de satellites de VEGA ? xxx
                                  (xxx est la réponse entrée)
```

```
= xxx
```

(PRINT <s1> ... <sN>) [FSUBR à N arguments]

édite dans le tampon de sortie les différentes S-expressions **<s1> ... <sN>** qui sont évaluées par la fonction PRINT puis imprime ce tampon (en faisant un appel implicite à la fonction (TERPRI 1)). PRINT retourne en valeur la valeur de l'évaluation de **<sN>**.

PRINT peut être défini en VLISP de la manière suivante :

```
(DF PRINT (l)
   (PROG1 (EVAL (CONS 'PRIN l)) (TERPRI 1)))
```

```
ex : ? (PRINT (1+ 9) (CDR '(A B C)))    ; forme à évaluer
       10 (B C)                         ; exécution
     = (B C)                            ; valeur retournée
```

(PRINCH <c> <n>) [SUBR à 2 arguments] ***{pour PRINt CHaracter}***

édite **<n>** fois le caractère **<c>**. Si <n> n'est pas un nombre ou est omis, le caractère **<c>** n'est édité qu'une fois. PRINC retourne **<c>** en valeur.

ex : (PRINCH " " 10) édite 10 caractères espace dans le tampon

```
(LET (n 4)
   (IF (= n 0)
      (EXIT)
      (PRINCH " " n)
      (PRINCH "*" (1+ (* (- 4 n) 2)))
      (TERPRI)
      (SELF (1- n))))
```

produira

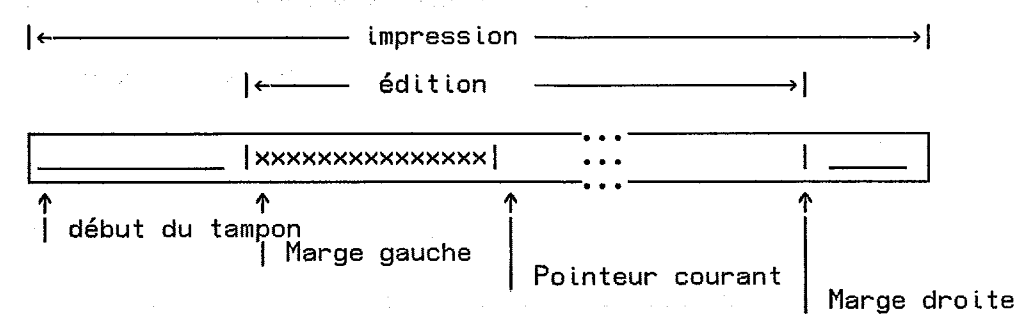
```
    *
   ***
  *****
 *******
```

4.4 CONTROLE DES FONCTIONS DE SORTIE

Les fonctions de sortie éditent les représentations externes des objets VLISP dans un tampon de sortie accessible au moyen de fonctions spécialisées.

4.4.1 Le tampon de sortie

Le tampon de sortie est organisé de la manière suivante :



Les éditions ne s'effectuent dans le tampon de sortie qu'entre la marge gauche et la marge droite alors que les impressions vont porter sur tout le tampon. L'accès aux valeurs de la marge gauche, du pointeur courant et de la marge droite est réalisé au moyen de fonctions spéciales ; il existe de plus une fonction OUTBUF permettant de manipuler directement n'importe quel caractère de ce tampon de sortie.

(LMARGIN <n>) [SUBR à 1 argument] *{pour Left MARGIN}*

permet de spécifier la marge à gauche **<n>** de l'impression. Par défaut à l'initialisation du système on a (LMARGIN 0). Si **<n>** n'est pas fourni, la marge n'est pas modifiée. LMARGIN retourne la valeur de la marge gauche courante et est principalement utilisée par le Pretty-print (Voir le chapitre 7) pour gérer automatiquement les renfoncements gauches à l'impression des structures de contrôle.

(RMARGIN <n>) [SUBR à 1 argument] *{pour Right MARGIN}*

permet de spécifier la marge à droite **<n>** de l'impression. Par défaut à l'initialisation du système on a (RMARGIN 62) pour les systèmes TRS80 et (RMARGIN 72) pour tous les autres systèmes. Si **<n>** n'est pas fourni, la marge n'est pas modifiée. RMARGIN retourne la valeur de la marge droite courante et est utilisée principalement pour régler la taille des lignes en fonction du terminal de sortie utilisé (télétype, écran, imprimante ...).

(OUTPOS <n>) [SUBR à 1 argument] *{pour OUTput POSition}*

permet de spécifier, si **<n>** est fourni et est un nombre, la nouvelle valeur du pointeur courant sur le tampon de sortie. Cet index pointe toujours sur le 1ère position libre dans le tampon. OUTPOS retourne en valeur la position courante de ce pointeur.

(OUTBUF <n> <c>) [SUBR à 2 arguments] *{pour OUTput tampon}*

considère le tampon de sortie comme un vecteur et permet donc d'avoir accès et/ou de modifier le **<n>**ième caractère du tampon de sortie avec le caractère **<c>**, si l'argument **<c>** est donné. OUTBUF retourne toujours la valeur du **<n>**ième caractère du tampon de sortie.

4.4.2 Limitations d'impression

Pour pouvoir limiter les impressions des structures très longues et rendre possible celles des listes circulaires deux nouvelles fonctions ont été ajoutées.

(PRINTLENGTH <n>) [SUBR à 1 argument]

permet de connaître et/ou de modifier (si l'argument numérique **<n>** est fourni) le nombre maximum d'éléments de liste qui est imprimé lors d'un PRINT ou d'un PRIN. Par défaut la valeur de ce nombre est de 2000. Si une liste contient un nombre d'éléments plus élevé, le reste des éléments ne sera pas imprimé et des points de suspension ... apparaitront devant la parenthèse fermante. Cette fonction permet de terminer l'impression des listes circulaires sur les CDR. PRINTLENGTH retourne en valeur la longeur maximum courante.

```
ex : (PRINTLENGTH 6)       ☞ 6
     '(1 2 3 4 5 6 7 8 9) ☞ (1 2 3 4 5 6...)
```

(PRINTLEVEL <n>) [SUBR à 1 argument]

permet de connaître et/ou de modifier (si l'argument numérique **<n>** est fourni) la profondeur maximum d'impression (i.e. le nombre maximum de parenthèses ouvrantes non fermées). En cas de dépassement, la liste qui provoque le débordement n'est pas imprimée et le caractère & est imprimé à sa place. Cette fonction permet d'imprimer des listes circulaires sur les CAR. PRINTLEVEL retourne en valeur la profondeur maximum courante.

```
ex : (PRINTLEVEL 3) ☞ 3

     '(DE FOO (L)
         (IF (NULL (CDR L)) L
             (FOO (CDR L))))
     ☞ (DE FOO (L) (IF (NULL &) L (FOO &)))

; impression des listes circulaires ;

?  (PRINTLEVEL 10)
=  10
?  (PRINTLENGTH 50)
=  50
```

```
? (SETQ L '(X Y Z))
= (X Y Z)
? (RPLACD (CDDR L) L)
= (Z X Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X
=  Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X...)
? (RPLACA L L)
= ((((((((((& Y Z & Y Z & Y Z & Y Z & Y Z & Y Z & Y Z &
= Y Z & Y Z & Y Z & Y Z & Y Z & Y Z & Y Z & Y Z & Y Z &
= Y...)...)...)...)...)...)...)...)...)...)
?
```

4.4.3 L'impression des listes circulaires ou partagées

L'utilisation des fonctions de limitation d'impression du paragraphe précédent, pour autant qu'elles stoppent l'impression, ne permettent pas de voir quels sont les doublets partagés. Pour cela il faut utiliser les fonctions suivantes écrites en VLISP qui utilisent la propriété de la fonction EQ (ici de la fonction MEMQ) suivante : (EQ l1 l2) est vraie si les 2 listes l1 et l2 sont les mêmes physiquement (ont la même adresse physique).

```
; Fonctions d'impressions circulaires simples :

(DE CPRINT (l)
   ; Fonction d'impression de l
   (CPRIN l)
   (TERPRI)
   'OK)

(DE CPRIN (l)
   ; Fonction d'édition de l
   (CPRIN1 l ()))

(DE CPRIN1 (l vus)
   ; Fonction auxillaire
   ; vus contient la liste des objets déjà visités
   (LET (l l)
      (IF (ATOM l)
         ; Les atomes peuvent être partagés
         (PROGN
            (STATUS PRINT #7)
            (PRIN l)
            (STATUS PRINT 0))
         ; En cas de liste
         (PRINCH "(")
         (WHILE (LISTP l)
            (IF (MEMQ l vus)
               ; C'est un objet déjà visité
               (EXIT (PRIN "...)"))
               ; C'est un objet nouveau
               (NEWL vus l)
               ; Récurse sur les CAR
               (SELF (NEXTL l))
               (IF l (PRINCH " "))))
         (IF (NULL l)
            ; La liste se termine bien
            ()
            ; C'est une paire pointée
            (PRINCH ".")
            (PRINCH " ")
            (PRIN l))
```

```
        ; Fermante finale
        (PRINCH ")")))))
```

ainsi si (SETQ LL '(A B C D))

l'appel (CPRINT (NCONC LL LL))

produira (A B C D ...)

4.4.4 Edition standard

Les différents objets VLISP sont édités dans le tampon de sortie en respectant l'unique régle suivante : la représentation externe d'un objet atomique ne peut être éditée à cheval sur deux lignes; ceci pour les atomes littéraux et les nombres.
L'impression de la fonction (QUOTE <s>) s'effectue '<s> (donc en restituant le macro-caractère standard d'entrée) pour des raisons de lisibilité.

Une fonction STATUS permet toutefois de régler certaines modalités d'impression et la fonction PTYPE permet de manipuler aisément le P-TYPE.

(STATUS PRINT <n>) [FSUBR]

permet de connaître (si la valeur **<n>** n'est pas fournie) ou de positionner (si la valeur **<n>** est fournie) les indicateurs internes de la fonction PRINT. Chaque indicateur est represénté par un bit dans la valeur **<n>**. A l'initialisation du système, tous ces indicateurs valent 0, il y a donc automatiquement un appel implicite de (STATUS PRINT 0).

Signification des bits du STATUS PRINT :

- bit 0 = 0 s'il faut insérer un espace avant chaque impression par PRIN ou PRINT; bit 0 = 1 s'il ne faut pas insérer d'espace avant chaque impression.

- bit 1 = 0 s'il ne faut pas faire précéder tous les caractères spéciaux des P-NAME par le caractère quote-caractère /. Si ce bit est à 1, il faut donc éditer le caractère / devant tout caractère spécial.

- bit 2 = 0 si les pseudo-chaînes sont restituées sans le caractère guillemets. Si ce bit est à 1 les pseudo-chaînes sont éditées encadrées du caractère délimiteur de chaîne, le ".

(PTYPE <at> <n>) [SUBR à 2 arguments]

permet de modifier (si l'argument **<n>** est fourni) ou de consulter (si l'argument **<n>** n'est pas fourni) la valeur du P-TYPE de l'atome littéral **<at>**. Ce P-TYPE est principalement utilisé par le PRETTY-PRINT pour y stocker le format à utiliser pour éditer cet atome en tant que fonction. Toutefois, le système gère le bit 7 qui est positionné si l'atome a été lu sous forme d'une pseudo-chaîne.
PTYPE retourne la valeur courante du P-TYPE de l'atome littéral **<at>**.

4.5 SELECTION DES FLUX D'ENTREE/SORTIE

Dans l'état minimum du système, VLISP 8 peut gérer simultanément
- un terminal d'entrée/sortie
- un terminal auxilliaire d'entrée/sortie

Certains systèmes VLISP 8 ont accès à un système de gestion de fichiers sur disquette :
- la version MDS a accès au système ISIS 1 ou 2
- la version TRS a accès aux systèmes TRSDOS, NEWDOS+ ou CP/M
- la version SORCERER a accès au système CP/M

Dans ces systèmes, VLISP 8 permet de manipuler en plus du terminal d'entrée/sortie et du terminal auxilliaire :
- un fichier d'entrée
- un fichier de sortie

Une spécification de fichier **<file>** est en VLISP 8 un atome littéral qui utilise la même syntaxe que le système hôte. Ces atomes contenant en général des caractères spéciaux, il est recommandé d'utiliser des pseudo-chaînes de caractères comme spécificateur de fichier.

```
sous ISIS     ":dd:ffffff.xxx "
sous TRSDOS   "ffffffff/xxx.pppppppp:d"
sous CP/M     "d:ffffffff.xxx"
```

dans lesquelles :
- dd.. est la spécification d'un périphérique
- ff.. est le nom du fichier
- xx.. est l'extension du fichier
- pp.. est la protection du fichier

```
ex : ":F1:PROG.VLI "
     ":LP: "
     "RESUL.LST "
     "PRETTY/VLI:1"
     "A:FOO.BAR"
```

ATTENTION : pour le système ISIS, la spécification doit TOUJOURS se terminer par un caractère espace.

Il est recommandé d'utiliser les extensions suivantes :
.VLI pour les fichiers source en VLISP 8
.LST pour les fichiers résultat

4.5.1 Les fonctions sur les flux

(INPUT <flux>) [SUBR à 1 argument]

cette fonction permet de sélecter le flux d'entrée **<flux>**. Ce flux peut être :
NIL, il s'agit alors du terminal.
T, il s'agit alors du périphérique auxilliaire.
<file>, il s'agit alors d'un fichier sur disque.

Si durant une lecture une erreur se produit, le message suivant apparait dans le flux de sortie :

*** erreur de lecture.* *ou*
*** read error.*

le fichier en question est fermé et le terminal est ouvert à sa place.

De même s'il n'est pas possible d'ouvrir le fichier **<file>** en entrée (le fichier n'existe pas, le périphérique n'est pas disponible ou pour toute autre raison ...) la fonction INPUT retourne NIL.

Enfin il est possible de déclencher avec cette fonction (ou avec les lectures qui vont suivre) des erreurs du système de gestion de disque lui-même.

(EOF) [SUBR à 0 argument] *{pour End Of File}*

cette fonction est automatiquement lancée par le système VLISP 8 à la fin du fichier d'entrée. EOF peut être également lancée par l'utilisateur pour provoquer artificiellement des fins de fichiers en lecture (le "SIGNAL EOF;" cher aux PL1istes). D'une manière standard EOF imprime le message d'avertissement suivant :

*** fin de fichier.* *ou*
*** E.O.F.*

puis effectue l'appel (INPUT) qui refait passer en mode terminal. EOF peut bien évidemment être redéfinie sous forme d'une EXPR pour gérer soi-même les fins de fichiers.

L'action standard de cette fonction peut être décrite en VLISP :

```
(DE EOF ()
   (PRINT "** Fin de fichier.")
   (INPUT))
```

Si cette fonction est redéfinie par l'utilisateur, elle doit comporter explicitement une ouverture d'un fichier d'entrée (au moyen de la fonction INPUT) sous peine de déclencher une erreur du système hôte.

Il n'y a jamais de fin de fichier sur le terminal en entrée sauf à définir un macro-caractère de fin de fichier. Par exemple :

```
   ; définition du caractère ↑Z (control-Z) comme fin de fichier d'entrée :

 ? (DMC "↑Z" () (EOF))
 = ↑Z
```

```
      ; la frappe d'un ↑Z provoquera l'appel de la fonction EOF

    ? ↑Z
      ** Fin de fichier
    ?
```

(OUTPUT <file>) [SUBR à 1 argument]

cette fonction permet de sélecter le flux de sortie **<flux>**. Ce flux peut être :
NIL, il s'agit alors du terminal.
T, il s'agit alors du périphérique auxilliaire.
<file>, il s'agit alors d'un fichier sur disque.

Si durant une écriture, une erreur se produit, le fichier en question est automatiquement fermé, le terminal est ouvert à sa place et le message suivant apparait :

*** erreur d'écriture.* *ou*
*** write error.*

De même s'il n'est pas possible d'ouvrir le fichier **<file>** en sortie (il n'y a plus de place, le périphérique n'est pas disponible ou pour toute autre raison ...) cette fonction retourne NIL.

Enfin il est possible de déclencher avec cette fonction (ou avec les écritures qui vont suivre) des erreurs du système de gestion de disque lui-même.

ex : de manipulation de fichiers

```
; fonction COPYFILE qui copie entièrement le fichier
; d'entrée <filin> dans le fichier de sortie <filout>

  (DE COPYFILE (<filin> <filout> ;; nbsexp)
       (OR (INPUT <filin>) (EXIT (LIST "Il n'y a pas : " <filin>)))
       (OR (OUTPUT <filout>) (EXIT (INPUT) (LIST "Problème avec : " <filout>)))
       (SETQ nbsexp 0)
       (STATUS PRINT #7)
       (ESCAPE EOF
          (WHILE T
             (SETQ nbsexp (1+ nbsexp))
             (PRINT (READ))))
       (INPUT)
       (OUTPUT)
       (STATUS PRINT 0)
       (PRINT "Copie de :" <filin> "sur" <filout>
           ".  " nbsexp "S-expressions écrites."))
       'OK))

 ? (COPYFILE "CIRC/SAV:0" "CP/VLI:1")
   COPIE DE CIRC/SAV:0 SUR CP/VLI:1  7 S-EXPRESSIONS LUES.
 = OK

 ? (COPYFILE "PRETTY.VLI " ":F1:PRETTY.VLI ")
   Copie de PRETTY.VLI sur :F1:PRETTY.VLI . 341 S-expressions écrites.
 = OK
```

On remarquera la redéfinition dynamique de la fonction EOF (au moyen d'un ESCAPE) qui permet de récupérer (à la manière d'une interruption) la condition fin de fichier.

4.5.2 Le fichier initial

A l'initialisation des systèmes VLISP 8 utilisant des fichiers, un ichier initial est lu avant de passer en mode conversationnel sur le terminal. Ce fichier se nomme :
- "VLISP.INI " dans les systèmes ISIS 1 et 2
- "VLISP/INI" dans les systèmes TRSDOS et NEWDOS+
- "VLISP.INI" dans les systèmes CP/M

Ce fichier contiendra les définitions de EXPR/FEXPR/MACRO que l'utilisateur désire posséder à chaque appel de VLISP 8.

Le texte de ce fichier de la version TRS est donné au paragraphe 8.1.

4.5.3 Le mode AUTOLOAD de fonctions

Comme il est fastidieux de charger à la main plusieurs fichiers de fonctions à chaque exécution VLISP 8 permet l'utilisation de ***fonctions autoloads*** (qui se chargent toutes seules dynamiquement à leur premier appel). Ceci est réalisé au moyen des 2 fonctions suivantes (qui peuvent bien entendu se trouver dans le fichier initial vu à la section précédente) :

```
(DF LIBRARY (l)
   ; chargement en douceur d'un fichier disque
   (IFN (INPUT (CAR l))
      (EXIT (LIST (CAR l) "n'existe pas."))
      (ESCAPE EOF
         (WHILE T
            (EVAL (READ))
            (IF (CADR l)(PRINCH "."))))
      (INPUT)
      (IF (CADR l) (TERPRI))
      (CAR l)))


(DF AUTOLOAD (l)
   ; définition de fonction autoload
   ; (AUTOLOAD fichier at1 ... atN)
   (MAPC (LAMBDA (at)
      (EVAL (LIST 'DM at '(l)
               (LIST 'FTYPE (LIST QUOTE at) 0)
               (LIST 'LIBRARY (CAR l))
               'l)))
     (CDR l)))
```

Ainsi l'appel :

```
(AUTOLOAD "PRETTY/VLI" PRETTY)
```

donne la définition suivante à la fonction PRETTY :

```
(DM PRETTY (l)
   (FTYPE 'PRETTY 0)
   (LIST 'LIBRARY "PRETTY/VLI")
```

l)

ce qui fait qu'à la première évaluation de (PRETTY ...) la MACRO est appelée. Cette MACRO se détruit elle-même (pour éviter de boucler au cas ou la fonction ne serait pas définie) puis évalue (LIBRARY "PRETTY/VLI") qui charge en silence le fichier "PRETTY/VLI".
La valeur retournée par la MACRO étant l'appel originel (PRETTY ...) lui-même, EVAL reévalue cette forme dans laquelle la fonction PRETTY est maintenant définie (cette nouvelle définition se trouvait dans le fichier). Ouf.



4.6 FONCTIONS SPECIALES SUR LE CASSETOPHONE

La plupart des systèmes TRS 80 peuvent utiliser un cassetophone. Son utilisation est la même qu'avec les autres processeurs du TRS80 (BASIC, EDTASM ...). Son fonctionnement sera donc erratique et générateur de nombreux tourments...

Il est possible de diriger les flux d'entrée et de sortie sur cassette, ce qui permet de ranger et de restaurer des définitions de fonctions sur cassette.

(INPUTAPE <i>) [SUBR à 1 argument]

si l'indicateur **<i>** est positionné (s'il est différent de NIL) permet d'associer la cassette au flux d'entrée. Toutes les lectures se feront donc sur la cassette. Si l'indicateur n'est pas positionné (si on évalue (INPUTAPE)) les lectures reprennent sur le flux d'entrée courant (i.e. sur le terminal). C'est l'évaluation d'une forme (INPUTAPE) qui provoque l'arrêt de la lecture sur cassette. Il est donc impératif que cette forme figure en fin de cassette.

(OUTPUTAPE <i>) [SUBR à 1 argument]

si l'indicateur **<i>** est positionné (s'il est différent de NIL) permet d'associer la cassette au flux de sortie. Toutes les impressions se feront donc sur la cassette. Si l'indicateur n'est pas positionné (si on évalue (OUTPUTAPE)) les impressions reprennent dans le flux de sortie courant (i.e. le terminal).

Voici une fonction VLISP capable d'enregistrer sur K7 une suite de définition de fonctions :

```
(DF FSAVE (l)
   ; l contient la liste des fonctions à écrire
   (OUTPUTAPE T)
   (WHILE l
      (PRINT
         (MCONS
            (CASSQ (FTYPE (CAR l))
               '((EXPR . DE) (FEXPR . DF) (MACRO . DM)))
            (CAR l)
            (FVAL (CAR l)))))
   (PRINT '(INPUTAPE ()))
   (OUTPUTAPE ())
   l)
```

(CSAVE) [SUBR à 0 argument]

permet de ranger sur K7 tout l'espace de travail de l'interprète. CSAVE retourne T en valeur.

(CLOAD) [SUBR à 0 argument]

permet de restorer tout l'espace de travail de l'interprète à partir de la K7. C'est donc l'opération inverse de la fonction CSAVE. CLOAD retourne T en valeur. Ces deux fonctions ne peuvent pas être écrite en VLISP.

4.7 FONCTIONS SPECIALES SUR TERMINAUX

Un ensemble de fonctions permettent de travailler directement sur le terminal (TTY) pour utiliser au mieux toutes ses possibilités. Ces fonctions n'utilisent pas le système d'entrées-sorties classique et peuvent donc être utilisées à tout moment.

4.7.1 Fonctions sur terminaux classiques

Ces fonctions sont utilisables sur n'importe quel terminal (en particulier sur tous les terminaux papier et à écran).

(TYI) [SUBR à 0 argument] *{pour TtY Input}*

retourne un nombre représentant le code interne du caractère suivant lu sur la TTY, sans aucune conversion et sans utiliser l'éditeur de ligne du moniteur (en particulier les caractères RUB-OUT, et les différents control-caractères ne sont plus effectifs).

(TYS) [SUBR à 0 argument] *{pour TtY Sneak}*

si un caractère est prêt à être lu sur le terminal, retourne la valeur que retournerait la fonction TYI, sinon si aucun caractère n'est prêt TYS retourne NIL. Cette fonction permet de tester si un caractère a été frappé sur le terminal. ATTENTION : si un caractère est effectivement retourné il est réellement lu.
Donc (UNTIL (TYS)) est bien équivalent à (TYI).

(TYO <n>) [SUBR à 1 argument] {pour TtY Output}

imprime sur le terminal le caractère de code interne **<n>**. Il n'y pas de conversion de caractère au moment de la sortie (par ex: les caractères contrôles sont envoyés directement sans passer par la forme ↑x). Il sera donc possible d'envoyer au moyen de cette fonction des caractères de contrôle (comme le déplacement du curseur) propre à chaque terminal.

4.7.2 Fonctions sur terminaux à écran

Ces fonctions ne sont disponibles que pour des terminaux à écran pour lesquels il est possible d'adresser le curseur. Actuellement seuls les systèmes TRS 80 possèdent ces fonctions. Le système LISP/LOGO de H. WERTZ (voir le chapitre 8) en fait une utilisation intensive.

(CLEAR <n>) [SUBR à 1 argument]

permet d'effacer tout l'écran avec le caractère de code interne **<n>** et de positionner le curseur au début de la première ligne. CLEAR retourne toujours T en valeur. Le code interne de l'espace étant le nombre #20, l'appel (CLEAR #20) efface donc tout l'écran.

(DISPLAY <n> <s>) [SUBR à 2 arguments]

cette fonction permet d'afficher sur un écran une suite de caractères à un emplacement donné. **<n>** est la position sur l'écran (qui est considéré comme un vecteur dont le premier élément est l'élément 0) à laquelle doit être affiché les caractères. Cet écran possède 16 lignes de 64 colonnes les positions sur l'écran variront donc de 0 à 1023. Si l'argument **<n>** n'est pas fourni, la position courante du curseur sur l'écran est utilisée.

L'argument **<s>** indique les caractères à afficher. Il peut être :

- un atome différent de NIL. Dans ce cas tous les caractères du P-NAME de cet atome sont affichés à la position indiquée par **<n>**. L'action est donc identique à la fonction PRIN.
- une liste de CODES INTERNES. DISPLAY va faire apparaître sur l'écran (à la position indiquée par **<n>**) cette suite de codes internes. Dans le système TRS 80 on a ainsi accès aux possiblités graphiques en utilisant en particulier les codes internes de 129 à 191 (ou de 80 à BF).
- rien (i.e. la valeur NIL ou l'absence d'argument). Dans ce cas aucun caractère n'est visualisé {on se demande ce qu'il pouvait faire [n.d.c]} Ce dernier type de l'argument **<s>** permet de changer la position du curseur sans rien faire apparaître de caractères sur l'écran.

DISPLAY retourne en valeur la position courante du curseur qui pointe sur le premier emplacement libre (après donc l'affichage).

La mémoire de rafraîchissement de l'écran dans le système TRS80 se trouve à l'adresse #3C00.

DISPLAY peut donc se traduire en VLISP de la manière suivante :

```
(DE DISPLAY (n l)
   (MAPC
      (LAMBDA (c)
         (MEMORY (+ #3C00 n) c)
         (SETQ n (1+ n)))
      l)
   n)
```

Voici une fonction qui montre les caractères graphiques du TRS 80 :

```
(DE GRAPH (n p pos cod)
   (CLEAR #20)
   (SETQ n 8 pos 0 cod 128)
   (WHILE (> n 0)
      (SETQ p 8)
      (WHILE (> p 0)
```

```
         (DISPLAY pos cod)
         (DISPLAY (+ pos 64) (LIST 45 COD 45))
         (SETQ p (1- p) cod (1+ cod) pos (+ pos 8)))
      (SETQ n (1- n) pos (+ pos 64)))
   (TYI))
```

l'appel étant réalisé :

```
(GRAPH)
```

(POINT <x> <y> <i>) [SUBR à 3 arguments]

permet d'allumer ou d'éteindre (en fonction de l'indicateur **<i>**) un point de l'écran semi-graphique du TRS 80. Cet écran semi-graphique se compose de 128 lignes (la position en **<x>**) et de 48 colonnes (la position en **<y>**). Les coordonnées débutent à la position 0.

```
ex : (POINT 5 5 T)   allume le point 5,5
     (POINT 5 6 T)   allume le point en dessous
     (POINT 6 6 T)   allume le point suivant à droite.
     (POINT 5 6)     efface le point 5 6
```

(WINDOW <n>) [SUBR à 1 argument]

permet de réserver une zone protégée sur l'écran. L'argument **<n>** est le nombre de lignes en haut de l'écran qu'il faut libérer. Cette zone est d'abord effacée par la fonction WINDOW puis le système ne l'utilise plus jamais (en particulier les ***scrollings*** du système préservent cette zone).
Cette fonction est utilisée pour réaliser des systèmes graphiques conversationnels.

A l'initialisation de VLISP 8, l'appel (WINDOW 2) est automatiquement réalisé ce qui permet de conserver les 2 premières lignes de l'écran. Pour pouvoir l'utiliser pleinement, évaluez (WINDOW 0).

Après évaluation de la fonction WINDOW, il n'est plus possible d'écrire sur les lignes réservées de l'écran, sauf à manipuler le curseur directement au moyen de la fonction DISPLAY.

Voici une fonction qui permet d'écrire un message sur la première ligne de l'écran, quelle que soit la position courante du curseur et l'état de l'écran.

```
(DE HEADER (l)
   (LET (ancienneposit (DISPLAY))
      (DISPLAY 0)
      (PRIN l)
      (DISPLAY ancienneposit)))
```

```
ex : (HEADER "J'y suis j'y reste.")
```

4.8 ENTREES/SORTIES SPECIALES

Le système VLISP 8 permet d'avoir accès au niveau le plus bas du système d'entrée/sortie des micro-processeurs en permettant d'utiliser les instructions IN et OUT. Pour les entrées/sorties utilisant le mode I/O mapped, la fonction d'accès à la mémoire (MEMORY) pourra être utilisée.

(IN <port>) [SUBR à 1 argument]

permet de lire la porte d'entrée d'adresse **<port>**, la valeur qui y est lue est retournée sous la forme d'un nombre de 8 bits.

(OUT <port> <val>) [SUBR à 2 arguments]

permet d'envoyer sur la porte de sortie d'adresse **<port>** les 8 bits de poids faibles du nombre **<val>**. OUT retourne en valeur le numéro de la porte utilisée.

ex : sur les systèmes TRS 80, l'évaluation de

```
(OUT #FF 8)
```

permet d'afficher sur l'écran des caractères en double largeur.

Si le terminal est connecté via un USART de type Intel 8251A d'adresse USART-STATUS et USART-DATA, les fonctions TYI, TYS, et TYO peuvent se décrire :

```
(DE TYI ()
   (WHILE (= 0 (LOGAND (IN USART-STATUS) 2)))
   ; lecture et écho
   (TYO (IN USART-DATA)))

(DE TYS ()
   (IF (= 0 (LOGAND (IN USART-STATUS) 2))
      NIL
      (IN USART-DATA)))

(DE TYO (n)
   (WHILE (= 0 (LOGAND (IN USART-STATUS) 1)))
   (OUT USART-DATA n)
   n)
```

4.9 LES SYSTEMES COLORIX

COLORIX est un système de visualisation couleur réalisé par Louis Audoire (1). Il existe 2 systèmes COLORIX :
- le système COLORIX 75
- le système COLORIX 79

Actuellement seuls certains systèmes (MZ80 et certaines versions TRS 80) peuvent être connectés aux systèmes COLORIX.

4.9.1 Le système COLORIX 75

Dans ce système, l'écran est considéré comme une matrice de 57 lignes et de 64 colonnes. Il est possible par programme de choisir une teinte (parmi 4096 combinaisons possibles) pour chacun des points de la matrice. Pour faciliter sa gestion, COLORIX 75 utilise un curseur courant interne sur le dernier élément modifié.

(ADRIX <n>) [SUBR à 1 argument]

permet de positionner le curseur interne de COLORIX 75 à la position absolue **<n>**. La matrice est dans ce cas considérée comme un vecteur de 64*57 éléments dont le 1er indice est 0. ADRIX retourne **<n>** en valeur.

(ADRIXY <x> <y>) [SUBR à 2 arguments]

permet de positionner le curseur interne de COLORIX 75 à la **<x>**ième ligne et à la **<y>**ième colonne. Rappelons que l'écran de COLORIX 75 utilise 57 lignes de 64 colonnes. ADRIXY retourne **<x>** en valeur.

ADRIXY peut être défini en VLISP de la manière suivante :

```
(DE ADRIXY (x y)
   (ADRIX (+ (* x 64) y)))
```

(COULIX <n>) [SUBR à 1 argument]

envoie sur COLORIX 75 à la position courante du curseur intenet, la valeur **<n>** considérée comme une couleur, puis le curseur interne est automatiquement incrémenté. Rappelons que les couleurs de COLORIX 75 se codent en binaire de la manière suivante :

0 0 0 0 r r r r b b b b v v v v

Il y a 4 bits pour coder chacunes des trois couleurs de base qui sont le rouge **<r>** le bleu **<b>** et le vert **<v>**, ce qui permet d'obtenir 4096 teintes différentes. Donc (COULIX #F00) envoie un point rouge, (COULIX #0F0) un point bleu, (COULIX #0FF) un point jaune ...

---

(1) voir ***COLORIX : un périphérique de visualisation couleur*** par Louis AUDOIRE, Mémoire de maîtrise, UER Informatique, Université de Paris 8 - Vincennes, Juin 1976.

COULIX retourne toujours la valeur <n> en valeur.

(COULDOSE <r> <b> <v>) [SUBR à 3 arguments]

envoie sur COLORIX 75 à la position du curseur interne, la couleur résultante du mélange des trois couleurs de base qui ont des intensitées comprises entre 0 et 15 (la valeur de chacune des couleurs de base est codée sur 4 bits de la même manière que pour la fonction COULIX), puis la position du curseur interne est automatiquement incrémentée. **<r>** représente l'intensité du rouge, **<b>** celle du bleu et **<v>** celle du vert. COULDOSE retourne **<r>** en valeur.

COULDOSE peut être défini en VLISP de la manière suivante :

```
(DE COULDOSE (r b v)
   (COULIX (+ (* r 256) (+ (* b 16) v))))
```

(COULT <n>) [SUBR à 1 argument]

initialise tout l'écran à la couleur **<n>**, puis le curseur interne est forcé à la position 0. La couleur **<n>** est codée d'une manière identique à fonction COULIX.

COULT peut être défini en VLISP de la manière suivante :

```
(DE COULT (coul)
   (ADRIX 0)                      ; positionnement en haut
   (LET (n (* 57 64))             ; prepare pour 57*64 points
      (IF (= n 0)                 ; le compte est bon ?
         ()                       ; oui : sort du LET
         (COULIX coul)            ; non : envoie un point
         (SELF (1- n))))          ;    et au suivant
   (ADRIX 0)                      ; repositionne en haut
   coul)                          ; retourne la couleur
```

4.9.2 Le système COLORIX 79

Le système COLORIX 79 est une version améliorée du système COLORIX 75. L'écran de COLORIX 79 est composé de 360 colonnes de 256 lignes chacunes. Chaque point de cet écran peut recevoir une couleur codée sur 12 bits (4 bits pour chaque couleur primaire). Il y a donc 4096 (2↑12) couleurs possibles. Dans toutes ces fonctions (de type FSUBR car les arguments sont évalués par les fonctions elles-mêmes), les arguments **<r>**, **<b>**, **<v>** représentent les 3 couleurs primaires (qui sont calculées modulo 16), les arguments **<x>**, **<y>** représentent des coordonnées (prises modulo 512).

(VISINI <r> <b> <v>) [FSUBR]

initialise tout l'écran à la couleur formée par la combinaison des 3 couleurs primaires **<r>**, **<b>**, **<v>**. VISINI retourne T en valeur.

```
ex : (VISINI)            efface l'écran
     (VISINI 15 15 15)   allume l'ecran
     (VISINI 15 0 0)     rempli l'écran de rouge.
```

(VISPOT <r> <b> <v> <x> <y>) [FSUBR]

force la couleur formée par la combinaison des 3 couleurs primaires **<r>**, **<b>**, **<v>**, dans le point de coordonnées **<x>** et **<y>**. VISPOT retourne T en valeur.

(VISEGM <r> <b> <v> <x1> <y1> ... <xN> <yN>) [FSUBR]

trace des segments de droite dans la couleur formée par la combinaison des 3 couleurs primaires **<r>**, **<b>**, **<v>**. Les coordonnées de ces segments de droite sont :
- x1 y1 ,, x2 y2
- x2 y2 ,, x3 y3
- xN-1 yN-1 ,, xN yN.

VISEGM trace donc une ligne polygonale monochrome.

pour dessiner un carré jaune au milieu de l'écran :

```
(VISEGM 0 15 15 10 10 40 10  40 40 10 40)
```

(VISMEM <x> <y>) [FSUBR]

retourne la couleur actuelle (sous la forme d'un nombre de 12 bits) du point de coordonnées **<x>** et **<y>**. Cette dernière fonction permet de lire la mémoire de rafraîchissement de COLORIX 79.




V - LES FONCTIONS SYSTEME

5.1 LE TOP-LEVEL

La boucle principale (ou TOP-LEVEL) de l'interprète consiste à évaluer indéfiniment la forme :

```
(WHILE T (TOPLEVEL))
```

C'est donc la fonction TOPLEVEL qui détermine le mode de fonctionnement de l'interprète. Cette fonction (de type SUBR) est bien évidemment redéfinissable par l'utilisateur qui désire se construire son propre système. Il existe de plus un STATUS qui permet de contrôler le fonctionnement de la fonction TOPLEVEL standard.

(TOPLEVEL) [SUBR à 0 argument]

Cette fonction va, d'une manière standard :
- lire une S-expression dans le flux d'entrée courant
- évaluer cette S-expression
- imprimer le résultat de cette évaluation dans le flux de sortie courant

(STATUS TOPLEVEL <n>) [FSUBR]

permet de connaître (si la valeur **<n>** n'est pas fournie) ou de positionner (si la valeur **<n>** est fournie) les indicateurs internes qui contrôlent la fonction standard TOPLEVEL. Chaque indicateur est représenté par un bit dans la valeur de **<n>**. A l'initialisation du système tous ces indicateurs valent 0, il y a donc automatiquement un appel implicite de :
    (STATUS TOPLEVEL 0)

Signification des bits du STATUS TOPLEVEL

- bit 0 = 0 s'il ne faut pas imprimer les formes lues qui doivent être évaluées. Si ce bit est à 1, toute forme lue est imprimée avant d'être évaluée.

- bit 1 = 0 s'il faut imprimer la valeur des évaluations au TOPLEVEL. Si ce bit est à 1, aucune valeur d'évaluation n'est imprimée.

à l'initialisation du système la fonction TOPLEVEL est équivalente à :

```
(DE TOPLEVEL () (PRINT (EVAL (READ))))
```

en tenant compte des bits du STATUS TOPLEVEL,
TOPLEVEL pourrait se définir en VLISP de la manière suivante :

```
(DE TOPLEVEL ( ;; s)
```

```
(SETQ s (READ))
(OR (= 0 (LOGAND (STATUS TOPLEVEL) 1) (PRINT s))
(SETQ s (EVAL s))
(AND (= 0 (LOGAND (STATUS TOPLEVEL) 2) (PRINT s)))
```

ATTENTION : une redéfinition de cette fonction, qui n'évalue pas des formes lues, rend tout le système VLISP inutilisable.

exemple de définition à éviter :

(DE TOPLEVEL () (READ) (PRINT 'MARRE))

5.2 LE GARBAGE-COLLECTING

La zone de l'interprète qui contient les doublets de liste est allouée dynamiquement. Quand cette zone est saturée une machinerie connue sous le nom de "Garbage-collecting" est automatiquement appelée pour récupérer les doublets de liste inutilisés.
Si cet essai s'avère infructueux, une erreur fatale se produit dont le libellé est :

*** erreur zone liste pleine.* ou
*** no room for list.*

De même s'il ne reste plus de place pour stocker les atomes littéraux, une erreur fatale se produit dont le libellé est :

*** erreur zone atome pleine.* ou
*** no room for atoms.*

(GC <i>) [SUBR à 1 argument]

permet d'appeler le Garbage-collecting explicitement. Si l'indicateur <i> est différent de NIL, le petit message suivant est édité dans le flux de sortie courant à la fin de la récupération :

NB GC : nnn ATOMES=xxxx LISTES=yyyy

dans lequel :
- nnn est le nombre de GC effectués depuis le début de la session
- xxxx est le nombre d'octets libres dans le zone atome
- yyyy est le nombre de doublets libres dans la zone liste.

Si avant ce message des nombres hexadécimaux sont imprimés dans le flux de sortie courant, cela indique que le G.C. a rencontré des OVNI (un OVNI étant comme chacun sait un Objet Vlisp Non Identifié). La rencontre de tels objets risque de perturber le fonctionnement de l'interprète par la suite.

Dans tous les cas GC retourne le nombre de doublets libres dans la zone liste.

5.3 ERREURS ET ARRET DE L'INTERPRETE

A l'apparition d'une erreur durant une évaluation, un message décrivant l'erreur (ainsi parfois qu'un argument défectueux) est édité dans le flux de sortie courant.
Tous ces messages commencent pas les caractères ***** erreur*** et ont été décris tout au long de ce manuel. Il existe deux formes pour chacun de ces messages : une forme française et une forme anglaise. La sélection de l'une de ces deux formes s'effectue durant l'assemblage de l'interprète.

En plus de ce message, plusieurs renseignements complémentaires sont également édités :

***** dernière forme : <s>*** ***ou***
***** last form : <s>***

qui indique la dernière forme qui a été évaluée par l'interprète.

Si l'erreur s'est produite à l'intérieur d'une fonction, le message suivant est également édité :

***** dernière fonction : <s>*** ***ou***
***** last function : <s>***

qui indique le corps de la dernière fonction appelée.

Dans tous les cas et avant de rentrer au top-level de VLISP, la fonction BREAK est appelée sans argument.

La valeur retournée par cette fonction est ensuite utilisée par l'interprète de la manière suivante :
Si cette valeur est égale à NIL (ce qui est la valeur retournée par défaut) il y a retour immédiat au top-level.
Si cette valeur est différente de NIL (par exemple égale à T) toutes les liaisons dynamiques des variables et des fonctions sont défaites.

(BREAK) [SUBR à 0 argument]

par défaut cette fonction ne fait qu'éditer dans le flux courant le message :

** BREAK

et retourne toujours la valeur NIL.

Cette fonction peut (et c'est tout son interêt) être redéfinie pour gérer soi-même les erreurs.

BREAK peut être définie en VLISP de la manière suivante :

```
(DE BREAK ()
   (PRINT "** BREAK")
   NIL)
```

mais peut également servir à réaliser un système plus complexe :

```
(SETQ BREAKNB 0)
```

```
(DE BREAK ()
   (SETQ BREAKNB (1+ BREAKNB))
   (PRINT "** BREAK  nb:" BREAKNB)
   (PRINT "Evalue (UNBREAK) pour en sortir.")
   (ESCAPE UNBREAK
      (WHILE T (TOPLEVEL)))
   (SETQ BREAKNB (1- BREAKNB))
   (IF (= BREAKNB 0)
      (PRINT "Je retourne au TOPLEVEL.")
      (PRINT "Je re-entre dans le BREAK  nb:" BREAKNB))
   (PRINT "Faut-il restaurer les variables")
   (TEREAD)
   (LET (rep (READCH))
      (COND
         ((CHRPOS rep "OTYoty")
            ; pour Oui Tak ou Yes
            (EXIT T))
         ((CHRPOS rep "Nn")
            ; pour Non Nie ou No
            (EXIT NIL))
         (T (PRINT "Répond O/N")
            (SELF (READCH))))))
```

(END) [SUBR à 0 argument]

arrête l'évaluation en cours, ferme tous les fichiers ouverts, sort de l'interprète, ne passe pas par la case départ et rend le contrôle au moniteur. END est utilisée pour sortir définitivement de l'interprète VLISP 8.2 et pour revenir au moniteur standard. END est la seule fonction standard qui ne retourne pas de valeur.



5.4 ACCES A LA MEMOIRE, AU C.P.U et au SYSTEME

Certaines fonctions permettent d'accéder directement à la mémoire pour pouvoir définir de nouvelles fonctions SUBR, pour tester l'interprète ... Ces fonctions servent à fabriquer le LAP VLISP 8.2 (le LAP est l'assembleur utilisable directement sous VLISP qui permet de charger le code produit par les compilateurs).

Ces fonctions utilisent des adresses mémoire quelconques. Les nombres VLISP ne pouvant pas représenter toutes les adresses possibles (du fait de la limitation des nombres entiers) une adresse mémoire **<adr>** sera représentée en VLISP de 2 manières possibles :
- un nombre
- une liste de 2 nombres de la forme : (<high> <low>)

dans laquelle le premier nombre <high> est l'octet de poids forts de l'adresse et le second <low> l'octet de poids faibles.

l'adresse #3C00 peut s'écrire également (#3C #00)

(LOC <s>) [SUBR à 1 argument] *{pour LOCation of}*

permet de connaître l'adresse de l'objet VLISP **<s**. L'adresse d'un objet VLISP est le pointeur sur la valeur de cet objet. Cette fonction permet de localiser en mémoire les objets VLISP 8 qui sont crés dynamiquement.

```
ex : (LOC '(A B C))      ☞ 15564
     (LOC '(A B C))      ☞ 13210
```

(MEMORY <adr> <n>) [SUBR à 2 arguments]

permet de consulter ou de modifier (si le 2ème argument numérique **<n>** est fourni) n'importe quel mot de la mémoire dont l'adresse **<adr>** est fournie en premier argument. Cette fonction ne fait AUCUN contrôle de validité d'adresse et doit donc être utilisée avec beaucoup de précautions. MEMORY retourne en valeur un nombre représentant la valeur de l'octet (après modification éventuelle) dont l'adresse est spécifiée.

(CALL <adr> <a1> <a2> <a3>) [SUBR à N arguments]

le premier argument **<adr>** doit être l'adresse d'un sous-programme en mémoire. CALL va lançer ce sous-programme après avoir charger les trois accumulateurs A1, A2 et A3 avec les valeurs respectives **<a1>**, **<a2> et <a3>** (i.e. les paires de registres HL, DE et BC). Cette fonction doit être utilisée avec précaution mais permet de réaliser l'interface machine-VLISP utilisée dans les compilateurs ou pour des traitements particuliers.
CALL retourne en valeur la valeur actuelle de la paire de registres HL après exécution du code débutant en **<adr>**.

(EXECUTE <adr> <l>) [SUBR à 2 arguments]

<adr> représente une adresse en mémoire. EXECUTE suppose que **<l>** est une liste de nombres représentant des INSTRUCTIONS. EXECUTE va charger cette liste d'intructions en mémoire à l'adresse spécifiée en premier argument et lui passer le contrôle au moyen de l'instruction machine CALL. Cette liste d'instruction doit OBLIGATOIREMENT se terminer par l'intruction RET (de code #C9).
Si la liste d'intructions (le deuxième argument) n'est pas fournie cette fonction transfert simplement le contrôle à l'adresse spécifiée comme avec la fonction CALL mais sans initialisation de registres.
EXECUTE retourne en valeur la valeur actuelle de la paire de registres HL après exécution du code.

ex : exécution d'un programme qui force tout l'écran
du système TRS 80 avec le code #F0 :

```
(EXECUTE  #43 #00         ;  ORG   /4300
   '( #21 #00 #3C         ;  LXI   HL,/3C00
      #36 #F0             ;  MVI   M,/F0
      #11 #01 #3C         ;  LXI   DE,/3C01
      #01 #FF #03         ;  LXI   BC,1023
      #ED #B0             ;  LDIR
      #C9                 ;  RET
    )
```

(SYSTEM) [SUBR à 0 argument]

retourne un atome littéral qui indique le nom du système VLISP 8 utilisé. Cet atome peut être :
- MDS pour les systèmes MDS
- TRS80 pour les systèmes TRS
- SORCERER pour les systèmes SORCERER

(DATE) [SUBR à 0 argument]

retourne la date du jour sous la forme d'une pseudo-chaîne de caractères.

ex : (DATE) ☞ 12-Feb-80

(TIME) [SUBR à 0 argument]

retourne l'heure du jour sous la forme d'une pseudo-chaîne de caractères.

ex : (TIME) ☞ 14:04:20

VI - L'EDITEUR EDITV

Les systèmes TRS 80 possèdent un éditeur temps réel sur écran vidéo, l'éditeur EDITV, qui permet de stocker sous forme de caractères et de corriger en mémoire centrale un programme source écrit en VLISP. Durant toute l'édition ce programme est visualisé sur l'écran vidéo du TRS 80.

La philosphie de cet éditeur est la suivante : le texte source est affiché en permanence sur l'écran ainsi qu'un curseur (qui est représenté par un petit rectangle blanc clignotant). Toute modification du texte source ou tout mouvement du curseur (ordonné au moyen de commandes entrées sur le clavier) est immédiatement visualisé ce qui permet de contrôler directement toutes les transformations effectuées au texte source.

Dans la version minimum de l'éditeur, il est possible de conserver une page entière (i.e. 16 lignes de 64 caractères) en mémoire. Cette page qui ne peut être modifiée que par l'éditeur peut être relue par VLISP et fait office de mémoire "bloc-note".

Il est notoire qu'une fois avoir utilisé ce type d'éditeur il devient difficile de travailler sans nervosité avec les éditeurs dits "aveugles" qui, affichant toujours les commandes et non les modifications réellement apportées au texte, demande un gros effort de concentration qu'il est difficile de soutenir longtemps.

Il est recommandé d'experimenter le maniement de cet éditeur au moyen d'une petite fonction de test car il est indispensable de ***voir*** l'action des commandes pour les comprendre et les utiliser pleinement.

L'éditeur possède différents mode de fonctionnement; i.e. qu'un caractère normal entré au clavier peut :

- remplacer le caractère qui se trouvait à l'emplacement du curseur. C'est le mode remplacement.

- être ajouté à l'emplacement du curseur après avoir décalé tous les autres caractères de la ligne. C'est le mode insertion.

- être traduit en caractère graphique (par ajout de la valeur #80) et remplacer le caractère qui se trouvait à l'emplacement du curseur. C'est le mode remplacement graphique.

A tout moment la position courante du curseur est visualisée par un petit carré blanc clignotant (s'il se trouve sur un caractère) ou fixe (s'il se trouve sur un espace).

6.1 LES CARACTERES DE COMMANDES

Un certain nombre de caractères sont interprétés par l'éditeur comme des caractères de commande (i.e. qu'ils ne sont pas inclus dans le texte source mais servent à commander des actions spécifiques de l'éditeur).

Ces caractères de commandes sont :
- certains caractères spéciaux se trouvant sur le clavier (comme les flèches ←↑→, la touche ENTER, la touche CLEAR ou la touche BREAK),
- tous les caractères SHIFT (i.e. les caractères normaux entrés avec la touche SHIFT enfoncée)

Ces caractères de commande sont regroupés en différentes classes :
- les commandes qui déplacent le curseur.
- les commandes qui demandent de rechercher une position dans la page.
- les commandes qui détruisent des caractères existants dans la page.
- les commandes qui permettent de changer de mode
- et un certain nombre de commandes de service de l'éditeur.

6.1.1 Commandes de déplacement du curseur

Ces commandes n'altèrent jamais le contenu de l'écran mais modifient la position du curseur.

← déplace le curseur d'une position à gauche

→ déplace le curseur d'une position à droite

↑ (flèche en haut) déplace le curseur d'une ligne vers le haut

v (flèche en bas) déplace le curseur d'une ligne vers le bas

SHIFT/→ déplace le curseur à la fin de la ligne courante

ENTER déplace le curseur au début de la ligne suivante

SHIFT/H déplace le curseur au début de la première ligne de l'écran

SHIFT/Z déplace le curseur sur le premier caractère libre en fin de page

6.1.2 Commande de recherche de position

Ces commandes ne modifient jamais le contenu de l'écran mais modifient la position du curseur. Si la recherche réussie, le curseur est déplacé sinon le curseur reste inchangé.

SHIFT/S puis <x> (<x> étant un caractère quelconque). Positionne le curseur sur le prochain caractère <x> rencontré dans la page. La recherche du caractère se fait à partir de la position courante du curseur jusqu'à la fin de la page, c'est donc une recherche en avant. S'il n'y a pas le caractère demandé le curseur ne bouge pas.

SHIFT/B puis <x> (<x> étant un caractère quelconque). Positionne le curseur sur le caractère <x> précédent de la page. La recherche du caractère se fait à partir de la position courante du curseur jusqu'au début de la page, c'est donc une recherche en arrière. Si le caractère demandé n'existe pas, le curseur ne bouge pas.

SHIFT/M si le curseur pointe sur une parenthèse ouvrante se positionne sur la parenthèse fermante correspondante. Si le curseur ne pointe pas directement sur une parenthèse ouvrante l'éditeur effectue automatiquement l'enchainement SHIFT/S (SHIFT/M. Cette commande permet de vérifier aisément un parenthésage

SHIFT/A permet de répéter la dernière commande de recherche de type SHIFT/B, SHIFT/S ou SHIFT/M. Cette commande évite la frappe du caractère à chercher en cas de recherches multiples.

6.1.3 Commandes de destruction de caractères existant

SHIFT/D détruit le caractère pointé par le curseur.

SHIFT/← correspond à l'enchaînemment : ← puis SHIFT/D

SHIFT/K détruit tous les caractères de la ligne courante à partir du pointeur courant.

CLEAR efface tout l'écran et positionne le curseur sur le premier caractère de la première ligne.

6.1.4 Commandes de manipulation de lignes

SHIFT/V (SHIFT flèche en bas) si le curseur se trouve en début de ligne permet de descendre d'une position toutes les lignes qui se trouvent sous le curseur. La dernière ligne de la page est perdue.

SHIFT/↑ (SHIFT flèche en haut) permet de remonter d'un position toutes les lignes qu se trouvent sous la curseur. La première ligne qui se trouvait sous le curseur est perdue et une nouvelle ligne vide se forme à la fin de la page. Cette commande défait donc la commande précédente.

6.1.5 Commandes de changement de mode

SHIFT/I passe en mode insertion. I.e. que tous les caractères normaux suivants seront insérés dans la ligne courante à partir de la position du curseur en déplaçant les caractères qui s'y trouvaient déjà. Ce mode reste jusqu'à l'apparition d'un caractère de commande de n'importe quel type.

SHIFT/N passe en mode remplacement (qui est le mode par défaut). Cette commande est en général utilisée pour sortir du mode insert ou graphique.

SHIFT/G passe en mode graphique, i.e. que tout caractère entré est converti en un équivalent graphique. Tout nouveau caractère de commande (par exemple SHIFT/N), fait repasser en mode normal (i.e. en mode remplacement).

SHIFT/Q puis <x> (<x> étant un caractère quelconque et même un caractère de contrôle). Permet de prendre le caractère <x> littéralement i.e. qu'il n'est jamais considéré comme caractère de commande. Par exemple SHIFT/Q suivi de ↑ fera apparaître sur l'écran le véritable caractère flèche en haut. SHIFT/Q est donc le quote-caractère de l'éditeur.

6.1.6 Commandes utilisant le cassetophone

Ces deux commandes permettent de sauver et de restorer le contenu de la page sur cassette.

SHIFT/R efface toute la page et lit une nouvelle page sur la cassette.

SHIFT/W permet de sauver toute la page sur cassette. Cette commande ne modifie ni l'écran ni la position du curseur.

6.1.7 Commandes de service

Elles permettent de sortir de l'éditeur EDITV.

SHIFT/E sauve tout l'écran dans la mémoire bloc-note et rend le contrôle au TOP-LEVEL de VLISP.

SHIFT/V sauve tout l'écran dans la mémoire bloc-note et rend le contrôle au TOP-LEVEL de VLISP qui doit relire la page éditée.

BREAK est identique à la commande SHIFT/E.



6.2 Appel de EDITV

(EDITV <n>) [SUBR à 1 argument]

appelle l'éditeur EDITV. Si l'argument **<n>** est égal à 1, la page précédement éditée (et qui a été sauvée en mémoire) est re-affichée sur l'écran et l'éditeur se place en mode remplacement, i.e. que tout caractère qui n'est pas un caractère de commande remplace le caractère pointé par le curseur.
Si l'argument **<n>** est égal à -1, c'est l'écran de VLISP, tel quel, qui est chargé dans la page éditeur. Cette nouvelle possibilité permet d'entrer quelques lignes sous VLISP et en cas d'erreur de copier l'écran de VLISP dans la mémoire bloc-note pour y faire des modifications ou bien de faire apparaître une fonction préalablement lue (par exemple au moyen de la fonction GETFN) puis de la rééditer.

Il existe un macro-caractère standard qui permet d'appeler directement l'éditeur : le caractère ***et commercial*** : &.

L'appel &1 est identique à (EDITV 1)
L'appel &-1 est identique à (EDITV -1)

Au retour de l'éditeur l'écran VLISP est restauré et EDITV retourne toujours la valeur T.

(EDIT <s>) [FEXPR]

permet d'éditer une fonction préalablement définie sous VLISP (si <s> est un atome littéral) ou bien la valeur d'une évaluation quelconque (si <s> n'est pas un atome littéral). Cette fonction n'est pas une FSUBR mais doit être décrite en VLISP sous forme de FEXPR de la manière suivante :

```
(DF EDIT (l)
   ; appel de (EDIT at)
   (WINDOW 0)            ; taille max de l'écran
   (CLEAR #20)           ; efface tout
   (LMARGIN 2)           ; aeration pour inclure
   (RMARGIN 60)          ;   les modifications
   (STATUS PRINT #7)     ; pour relire les impressions
   (TERPRI)              ; 1ère ligne vide
   (PRINT
      (IF (LITATOM l)
         (GETFN l)       ; la fonction nommée
         (EVAL (CAR l))));  l'expression évaluée
   (STATUS PRINT #0)     ; impression normale
   (DISPLAY () '(#20))   ; efface le curseur
   (EDITV -1))           ; on y va ...
```

Si cette fonction existe, il est possibl de redéfinir le macro-caractère "_", pour le généraliser, de la manière suivante :

```
(DMC "&" ()
   (LET (l (READ))
      (LIST (IF (NUMBP l) 'EDITV 'EDIT) l)))
```

Récapitulatif des commandes de EDITV

↑	monte d'une ligne
→	avance d'un caractère
v	descend d'une ligne
←	recule d'un caractère
SHIFT/↑	détruit la ligne suivante
SHIFT/→	positionnement en fin de la ligne courante
SHIFT/v	libère une nouvelle ligne
SHIFT/←	recule et détruit le caractère courant
ENTER	change de ligne
BREAK	sort de l'éditeur (identique à SHIFT/E)
CLEAR	efface tout l'écran
SHIFT/A	répète la dernière commande (S/B/M/Q)
SHIFT/B	<x> cherche le caractère <x> en arrière
SHIFT/C	--- rien ---
SHIFT/D	détruit le caractère courant
SHIFT/E	sort de l'éditeur
SHIFT/F	--- rien ---
SHIFT/G	entre en mode graphique
SHIFT/H	positionnement en haut de la page
SHIFT/I	entre en mode insertion
SHIFT/J	--- rien ---
SHIFT/K	détruit la fin de la ligne
SHIFT/L	--- rien ---
SHIFT/M	cherche la parenthèse fermante
SHIFT/N	repasse en mode remplacement
SHIFT/O	--- rien ---
SHIFT/P	--- rien ---
SHIFT/Q	<x> quote le caractère suivant <x>
SHIFT/R	lit une nouvelle page sur la K7
SHIFT/S	<x> cherche le caractère <x> en avant
SHIFT/T	--- rien ---
SHIFT/U	--- rien ---
SHIFT/V	rappelle VLISP
SHIFT/W	sauve la page sur la K7
SHIFT/X	--- rien ---
SHIFT/Y	--- rien ---
SHIFT/Z	passe en fin d'écran

VII - LE PRETTY-PRINT

Les fonctions standards PRINT et PRIN sont d'ordinaire utilisées pour éditer les S-expressions VLISP. Les seules mesures prises pour améliorer la lisibilité sont :
- l'insertion d'un espace entre chaque atome ;
- l'interdiction d'éditer un atome (atome littéral, nombre ou pseudo-chaîne de caractères) sur deux lignes.

Ces mesures sont nettement insuffisantes pour éditer vos programmes. Les fonctions du Pretty-Print vont les éditer d'une manière beaucoup plus lisible en faisant resortir, au moyen de renfoncements gauches et de sauts de lignes ad hoc, la structure de contrôle de vos fonctions.

Pour améliorer la lisibilité, tous les appels de la fonction QUOTE sont transcrits en utilisant le macro-caractère ' .

Les fonctions qui vont être décrites se trouvent dans un fichier disque dont le nom est :
- "PRETTY.VLI " sous système ISIS
- "PRETTY/VLI" sous système TRSDOS
- "PRETTY.VLI" sous système CP/M

Ce fichier doit être chargé avant utilisation de ces fonctions au moyen de l'appel :

```
(INPUT nom-du-fichier)
```

Une autre possibilité consiste à définir sous forme de fonctions AUTOLOAD les fonctions du PRETTY-PRINT, au moyen des formes

```
(AUTOLOAD "PRETTY.VLI" PRETTY)
(AUTOLOAD "PRETTY.VLI" PRETTYF)
(AUTOLOAD "PRETTY.VLI" SAVEF)
```

dans le fichier initial.

7.1 Les fonctions du PRETTY-PRINT

(PPRINT <s>) FEXPR

pretty-print l'expression **<s>** dans le flux de sortie courant. PPRINT retourne **<s>** en valeur.

(PRETTY <at>) FEXPR

<at> doit être le nom d'un atome qui possède une définition de fonction. PRETTY va éditer cette fonction dans le flux de sortie courant et retourner **<at>** en valeur.

(PREDIT <at>) FEXPR

<at> doit être le nom d'un atome qui possède une définition de fonction. PREDIT permet d'éditer le texte de cette fonction avec l'éditeur cablé EDITV. Cette fonction n'est utilisable qu'avec les systèmes possédant ce type d'éditeur cablé.

7.2 Les formats du PRETTY-PRINT

Le PRETTY-PRINT utilise les formats suivants :

Format 1 : de type PROGN

Format 2 : de type LAMBDA

Format 3 : de type DE

Format 4 : de type COND

Format 5 : de type SELECTQ

Format 6 : de type SETQ

Ces différent formats sont rangés dans le P-type des atomes littéraux. L'accès à ce P-type est réalisé au moyen de la fonction standard PTYPE.



7.3 Le texte du PRETTY-PRINT

Voici le texte complet de ces fonctions du PRETTY-PRINT :

```
;                P R E T T Y  -  P R I N T                  ;
;                                                           ;
;               Paragrapheur   VLISP 8 . 2                  ;
;                 ( Taille = 450 doublets )                 ;
;-----------------------------------------------------------;
;                Jérôme CHAILLOUX                           ;
;                                                           ;
;       Département d'Informatique                          ;
;       Université de Paris VIII - Vincennes                ;
;       Route de la Tourelle 75571 Paris Cédex 12           ;
;       Tél : 374 12 50 poste 299                           ;
;                                                           ;
;       I.R.C.A.M.                                          ;
;       31 Rue St Merri 75004 Paris                         ;
;       Tél : 277 12 33 poste 48-48                         ;
;-----------------------------------------------------------;

(STATUS TOPLEVEL 2)  ; Silence.

;----- Fonctions internes

(DE P-P (l)
   ; Pretty-Print l'expression l
   (COND
      ((NULL l) (PRINCH "(") (PRINCH ")"))
      ((ATOM l) (PRIN l))
      ((AND (EQ (CAR l) QUOTE) (NULL (CDDR l)))
         (PRINCH "'")
         (P-P (CADR L)))
      (T (PRINCH "(")
         (P-P (CAR l))
         (SELECTQ (PTYPE (NEXTL l))
            (1 ; Format PROGN
               (P-PROGN))
            (2 ; Format WHILE
               (P-P1) (P-PROGN T))
            (3 ; Format DEF
               (P-P1) (P-P1) (P-PROGN T))
            (4 ; Format COND
               (P-COND))
            (5 ; Format SELECTQ
               (P-P1) (P-COND))
            (6 ; Format SETQ multiple
               (T+3)
               (WHILE l (P-P1) (P-P1) (IF l (TERPRI)))
               (T-3))
            (  ; Format standard
               (T+3)
               (WHILE (LISTP l) (P-P1))
               (T-3)))
           (AND l (PRINCH " ") (PRINCH ".") (PRINCH " ") (PRIN l))
           (PRINCH ")"))))
```

```
(DE P-P1 ()
   ; Pretty Print un élément de la liste l
   ; mais pas le 1er !
   (PRINCH " ")
   (P-P (NEXTL l)))


(DE P-PROGN (?)
   ; Pretty Print le PROGN l si ?=T il indente toujours
  (IF (AND (NULL (CDR l)) (NULL ?))
       ; 1 seul argument
       (P-P1)
       ; plusieurs arguments
       (T+3)
       (WHILE (LISTP l)
          (TERPRI)
          (P-P (NEXTL l)))
       (T-3)))


(DE P-COND ()
   ; Pretty Print le COND l
   (T+3)
   (WHILE (LISTP l)
      (TERPRI)
      (PRINCH "(")
      (LET (l (NEXTL l))
         (P-P (NEXTL l))
         (IF l (P-PROGN)))
      (PRINCH ")"))
   (T-3))


(DE T+3 ()
   ; effectue un renfoncement droit
   (LMARGIN (+ (LMARGIN) 3)))


(DE T-3 ()
   ; enlève un renfoncement droit
   (LMARGIN (- (LMARGIN) 3)))


;*****  Fonctions standard du PRETTY PRINT

(DE PPRINT (s)
   ; Pretty Print l'expression s
   (STATUS PRINT #7)
   (LMARGIN 0)
   (P-P s)
   (TERPRI)
   (STATUS PRINT #0)
   s)


(DF PRETTY (l)
   ; Pretty Print la fonction (CAR l)
   ; ex: (PRETTY PRETTY)
   (PPRINT (GETFN (CAR l)))
   (CAR l))
```

```
(DF PREDIT (l)
   ; Edite la fonction (CAR l) pretty-printée
   ; est équivalent à (EDIT l)
   (WINDOW 0)
   (CLEAR #20)
   (LMARGIN 2)
   (RMARGIN 60)
   (STATUS PRINT #7)
   (TERPRI)
   (PPRINT (IF (LITATOM (CAR l)) (GETFN (CAR l)) (EVAL (CAR l))))
   (LMARGIN 0)
   (STATUS PRINT 0)
   (DISPLAY () '(#20))
   (EDITV -1))

;----- Fonction de sauvetage du fichier

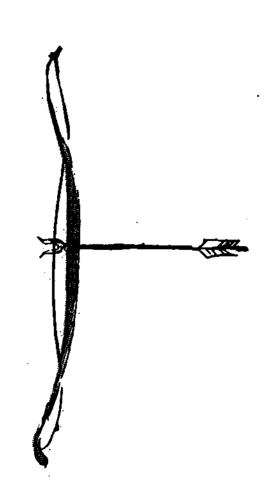
(DE SAVEF-PRETTY ()
   ; Pour tout sauver sur disquette
   (SAVEF "PRETTY/SAV"
        (STATUS TOPLEVEL 2)
        P-P P-P1 P-PROGN P-COND T+3 T-3
        PPRINT PRETTY PREDIT SAVEF-PRETTY
        (PROGN (STATUS TOPLEVEL 0) "J'ai charge PRETTY/VLI")))

;----- Epilogue standard

(PROGN
   (STATUS TOPLEVEL 0)
   "J'ai charge PRETTY/VLI")
```







VIII - La PROGRAMMATION EN VLISP 8

Ce chapitre fourni un certain nombre de programmes de démonstration écrits en VLISP 8.2.

8.1 Le fichier initial du système VLISP TRS

```
;              V L I S P   /   I N I                       ;
;                                                          ;
;           Fichier initial  VLISP 8 . 2                   ;
;              ( Taille < 300 doublets )                   ;
;----------------------------------------------------------;
;              Jérôme CHAILLOUX                            ;
;                                                          ;
;         Département d'Informatique                       ;
;         Université de Paris VIII - Vincennes             ;
;         Route de la Tourelle 75571 Paris Cédex 12        ;
;         Tél : 374 12 50 poste 299                        ;
;                                                          ;
;         I.R.C.A.M.                                       ;
;         31 Rue St Merri 75004 Paris                      ;
;         Tél : 277 12 33 poste 48-48                      ;
;----------------------------------------------------------;

(STATUS TOPLEVEL 2)  ; Silence.

;----- la MACRO LET

(DM LET (l)
   (RPLACB l
      (COND
         ((NULL (CADR l ))
            (CONS (CONS LAMBDA (CONS () (CDDR l )))))
         ((ATOM (CAADR l ))
            (CONS (CONS LAMBDA (CONS (LIST (CAADR l )) (CDDR l )))
                (CDADR l )))
         (T (CONS  (CONS LAMBDA (CONS (MAPCAR 'CAR (CADR l )) (CDDR l )))
                   (MAPCAR 'CADR (CADR l )))))))

;----- Appel de l'editeur interne

(DF EDIT (l)
   ; édite une fonction ou une expression
   ; ex : (EDIT FOO)   sans quote !
   (WINDOW 0)
   (CLEAR #20)
   (LMARGIN 2)
   (RMARGIN 60)
```

```
   (STATUS PRINT #7)
   (PRINT
      (IF (LITATOM (CAR l))
         (GETFN (CAR l))
         (EVAL (CAR l))))
   (STATUS PRINT #0)
   (DISPLAY () '(#20))
   (EDITV -1))


;-----  Sauvetage sur disque de fonctions

(DF SAVEF (l)
   ; sauvegarde de fonctions sur disque
   ; ex d'appel :
   ; (SAVEF file at1 ... atN)  sans quote!
   (OUTPUT (NEXTL l))
   (STATUS PRINT #7)
   (LMARGIN 0)
   (WHILE l
      (PRINT
         (IF (LITATOM (CAR l))
            (GETFN (NEXTL l))
            (NEXTL l)))
      (TERPRI))
   (STATUS PRINT #0)
   (OUTPUT))

(DMC "&" ()
   ; le macro-caractère correspondant
   (LET (l (READ))
      (LIST (IF (NUMBP l) 'EDITV 'EDIT) l)))


;-----  les fonctions qui manquent

(DF MCONS (l)
   (CONS (EVAL (CAR l))
      (EVAL (IF (NULL (CDDR l))
                (CADR l)
                (LIST 'MCONS (CDDR l))))))


 (DE MAPCAR (-f -l)
    (IF (NULL -l)
       ()
       (CONS (-f (CAR -l)) (MAPCAR -f (CDR -l)))))


 (DE MAPC (-f -l)
    (IF (NULL -l)
       ()
       (-f (CAR -l))
       (MAPC -f (CDR -l))))


;----- LIBRARY et AUTOLOAD

(DF LIBRARY (l)
   ; chargement en douceur d'un fichier disque
   (IFN (INPUT (CAR l))
      (EXIT (LIST (CAR l) "n'existe pas."))
      (ESCAPE EOF (WHILE T (EVAL (READ))))
```

```
      (INPUT)
      (CAR l)))


(DF AUTOLOAD (l)
   ; définition de fonction autoload
   ; (AUTOLOAD fichier at1 ... atN)
   (MAPC (LAMBDA (a)
      (EVAL
         (LIST 'DM a '(l)
             (LIST 'FTYPE (LIST QUOTE a) 0)
             (LIST 'LIBRARY (CAR l)) 'l)))
      (CDR l)))


(AUTOLOAD "ELEPH/VLI" VOIR)
(AUTOLOAD "HANOI/VLI" HANOI)
(AUTOLOAD "PRETTY/VLI" PRETTY PREDIT)
(AUTOLOAD "TRACE/VLI" TRACE UNTRACE)
(AUTOLOAD "LOGO/VLI" ER)


;----- Visualisation d'un fichier disque

(DE TYPE (l)
   (OR (INPUT l) (EXIT (LIST l "n'existe pas.")))
   (ESCAPE EOF
      (WHILE T (IF (TYS) (EOF) (PRINCH (READCH)))))
   (INPUT)
   l)


;----- C'est toujours utile

(DE FOO (l) (IF (NULL (CDR l)) l (FOO (CDR l))))


;----- Pour le sauvetage du fichier

(DE SAVEF-VLISP ()
   ; sauvetage de tout le fichier
   (SAVEF "VLISP/SAV"
      (STATUS TOPLEVEL 2)
      LET EDIT
      (DMC "&" ()
         (LET (l (READ))
            (LIST (IF (NUMBP l) 'EDITV 'EDIT) l)))
      SAVEF MCONS MAPCAR MAPC
      LIBRARY AUTOLOAD
      (AUTOLOAD "ELEPH/VLI" VOIR)
      (AUTOLOAD "HANOI/VLI" HANOI)
      (AUTOLOAD "PRETTY/VLI" PRETTY PREDIT)
      (AUTOLOAD "TRACE/VLI" TRACE UNTRACE)
      (AUTOLOAD "LOGO/VLI" ER)
      TYPE FOO SAVEF-VLISP
      (PROGN (STATUS TOPLEVEL 0)
         (LMARGIN 0) (RMARGIN 60) "J'ai charge VLISP/INI")))

;----- Final de l'initialisation

(PROGN
   (STATUS TOPLEVEL 0)
   (LMARGIN 0)
```

```
   (RMARGIN 60)
   "J'ai charge VLISP/INI")

;;â
```

8.2 Les fonctions de TRACE

```
;                 T R A C E   .   V L 8                  ;
;                                                        ;
;                  traceur   VLISP 8 . 2                 ;
;                ( Taille = 450 doublets )               ;
;--------------------------------------------------------;
;               Jérôme CHAILLOUX                         ;
;                                                        ;
;       Département d'Informatique                       ;
;       Université de Paris VIII - Vincennes             ;
;       Route de la Tourelle 75571 Paris Cédex 12        ;
;       Tél : 374 12 50 poste 299                        ;
;                                                        ;
;       I.R.C.A.M.                                       ;
;       31 Rue St Merri 75004 Paris                      ;
;       Tél : 277 12 33 poste 48-48                      ;
;--------------------------------------------------------;

(STATUS TOPLEVEL 2)  ; Silence.

;----- Fonctions de trace (elles sont AUTOLOAD)

(DF TRACE (l)
   ; trace la liste de fonctions l
   ; ces fonctions ne doivent pas contenir d'appels
   ; de fonctions SELF ou  EXIT
   (MAPC (LAMBDA (f)
      (LET (fval (FVAL f))
         (IF (ATOM fval)
            (PRINT "C'est quoi ?" f)
            (PUT f fval 'TRACE)
            (FVAL f
               (LIST (CAR fval)
                  (LIST 'PRINT
                     (LIST QUOTE f)
                     (LIST QUOTE '--->)
                     (LIST (IF (ATOM (CAR fval)) 'EVAL 'EVLIS)
                         (LIST QUOTE (CAR fval))))
                  (LIST 'LET
                     (LIST 'l (CONS 'PROGN (CDR fval))
                     (LIST 'PRINT
                        (LIST QUOTE f)
                        (LIST QUOTE '<---)
                        'l))))))))
      l)
   l)
```

```
(DF UNTRACE (l)
   ; enleve la trace de la liste des fonctions l
   (MAPC (LAMBDA (f)
      (LET (fval (GET f 'TRACE))
         (IF (ATOM fval)
            (PRINT f "n'etait pas tracee ...")
            (FVAL f fval)
            (REMPROP f 'TRACE)))
        l)
   l)

;----- Pour sauver tout le fichier

(DE SAVEF-TRACE ()
   ; sauve tout le fichier
   (SAVEF "TRACE/SAV"
      (STATUS TOPLEVEL 2)
      TRACE UNTRACE SAVEF-TRACE
      (PROGN (STATUS TOPLEVEL 0) "J'ai charge TRACE/VLI")))

;----- Epilogue standard

(PROGN
   (STATUS TOPLEVEL 0)
   "J'ai charge TRACE/VLI")

;;â
```

8.3 Les tours de Hanoi animées

Ce programme est une version ***animée*** du très célèbre problème des tours de Hanoi. Il est conçu pour fonctionner sur un système TRS80 et utilise les fonctions pseudo-graphiques de ce système. Voici le texte de ce programme :

```
;              H A N O I   .   V L 8                  ;
;                                                     ;
;       Les Tours de HANOI animées en  VLISP 8 . 2    ;
;            ( Taille 800 doublets )                  ;
;-----------------------------------------------------;
;              Jérôme CHAILLOUX                       ;
;                                                     ;
;       Département d'Informatique                    ;
;       Université de Paris VIII - Vincennes          ;
;       Route de la Tourelle 75571 Paris Cédex 12     ;
;       Tél : 374 12 50 poste 299                     ;
;                                                     ;
;       I.R.C.A.M.                                    ;
;       31 Rue St Merri 75004 Paris                   ;
;       Tél : 277 12 33 poste 48-48                   ;
;-----------------------------------------------------;
```

```
(STATUS TOPLEVEL 2)   ; silence.

; dessin des aiguilles et des disques

(DE init-disque ()
   (SETQ
   l-disque  '(
        (#20 #20 #20 #20 #20 #20 #20 #20 #20 #FF
         #20 #20 #20 #20 #20 #20 #20 #20 #20)
        (#20 #20 #20 #20 #20 #20 #20 #20 #F0 #FF
         #F0 #20 #20 #20 #20 #20 #20 #20 #20)
        (#20 #20 #20 #20 #20 #20 #20 #F0 #F0 #FF
         #F0 #F0 #20 #20 #20 #20 #20 #20 #20)
        (#20 #20 #20 #20 #20 #20 #F0 #F0 #F0 #FF
         #F0 #F0 #F0 #20 #20 #20 #20 #20 #20)
        (#20 #20 #20 #20 #20 #F0 #F0 #F0 #F0 #FF
         #F0 #F0 #F0 #F0 #20 #20 #20 #20 #20)
        (#20 #20 #20 #20 #F0 #F0 #F0 #F0 #F0 #FF
         #F0 #F0 #F0 #F0 #F0 #20 #20 #20 #20)
        (#20 #20 #20 #F0 #F0 #F0 #F0 #F0 #F0 #FF
         #F0 #F0 #F0 #F0 #F0 #F0 #20 #20 #20)
        (#20 #20 #F0 #F0 #F0 #F0 #F0 #F0 #F0 #FF
         #F0 #F0 #F0 #F0 #F0 #F0 #F0 #20 #20)
        (#20 #F0 #F0 #F0 #F0 #F0 #F0 #F0 #F0 #FF
         #F0 #F0 #F0 #F0 #F0 #F0 #F0 #F0 #20)
        (#F0 #F0 #F0 #F0 #F0 #F0 #F0 #F0 #F0 #FF
         #F0 #F0 #F0 #F0 #F0 #F0 #F0 #F0 #F0)
        (#FF #FF #FF #FF #FF #FF #FF #FF #FF #FF
         #FF #FF #FF #FF #FF #FF #FF #FF #FF))))

;----- les fonctions de mouvements de disques

(DE monte (n pos nb ;; d e)
   ; monte le disque de numero n en pos de nb places
   (SETQ pos (+ pos (* 64 (+ nb 3)))
         d (CNTH (1+ n) l-disque)
         e (CNTH 1 l-disque))
   (WHILE (>= nb 0)
      (DISPLAY (- pos 64) d)
      (DISPLAY pos e)
      (SETQ pos (- pos 64) nb (1- nb))))

(DE descend (n pos nb ;; d e)
   ; descend le disque de numero n en pos dè nb places
   (SETQ pos (+ pos 128) d (CNTH (1+ n) l-disque) e (CNTH 1 l-disque))
   (WHILE (> nb 0)
      (DISPLAY (+ pos 64) d)
      (DISPLAY pos e)
      (SETQ pos (+ pos 64) nb (1- nb))))

(DE avance (n dep arr ;; pos nb d e)
   ; avance le disque de numero N de DEP vers ARR
   (SETQ pos (+ dep 128)
         nb (- arr dep)
         d (CNTH (1+ n) l-disque)
         e (CNTH 1 l-disque))
   (IF (> nb)
```

```
    (WHILE (> nb)
       (DISPLAY pos e)
       (DISPLAY (1+ pos) d)
       (SETQ pos (1+ pos) nb (1- nb)))
    (WHILE (<= nb)
       (DISPLAY pos e)
       (DISPLAY (1- pos) d)
       (SETQ pos (1- pos) nb (1+ nb)))))


(DE bouge (n depart arrivee)
   ; realise un mouvement
   (monte n (GET depart 'pos) (- 10 (LENGTH (CVAL depart))))
   (avance n (GET depart 'pos) (GET arrivee 'pos))
   (descend n (GET arrivee 'pos) (- 10 (LENGTH (CVAL arrivee))))
   (SET depart (CDR (CVAL depart)))
   (SET arrivee (CONS n (CVAL arrivee))))


(DE affiche (l pos ;; n)
   ; affiche le contenu de toute une aiguille
   (SETQ pos (+ pos 768) n 11)
   (WHILE (> n 0)
        (DISPLAY pos (CNTH (IF l (1+ (NEXTL l)) 1) l-disque))
        (SETQ pos (- pos 64) n (1- n))))


(DE Hanoi-roule (n depart arrivee intermediaire)
   ; le moteur de HANOI
   (IF (<= n 0)
        ()
        (Hanoi-roule (1- n) depart intermediaire arrivee)
        (bouge n depart arrivee)
        (Hanoi-roule (1- n) intermediaire arrivee depart)))


(DE Hanoi (n)
   ; le lanceur de HANOI
   (CLEAR #20)
   (init-disque)
   (DISPLAY 20 "LES TOURS DE HANOI")
   (DISPLAY 980 "Combien de disques ")
   (SETQ n (READ)
         aiguille-A '(9 10)
         aiguille-B '(9 10)
         aiguille-C '(9 10))
   (PUT 'aiguille-A 0 'pos)
   (PUT 'aiguille-B 21 'pos)
   (PUT 'aiguille-C 42 'pos)
   (IF (OR (< n 3) (> n 8)) (Hanoi))
   (LET (n n) (IF (= n 0) () (NEWL aiguille-A n) (SELF (1- n))))
   (affiche (REVERSE aiguille-A) (GET 'aiguille-A 'pos))
   (affiche (REVERSE aiguille-B) (GET 'aiguille-B 'pos))
   (affiche (REVERSE aiguille-C) (GET 'aiguille-C 'pos))
   (Hanoi-roule n 'aiguille-A 'aiguille-B 'aiguille-C))


(DE savef-Hanoi ()
   ; pour sauver tout ce fichier
   (SAVEF "HANOI/SAV"
      (STATUS TOPLEVEL 2)
      init-disque monte descend avance bouge
```

```
      Hanoi-roule affiche Hanoi savef-Hanoi
      (PROGN
         (STATUS TOPLEVEL 0)
         "(HANOI) pour lancer.")))


(PROGN
   (STATUS TOPLEVEL 0)
   "(HANOI) pour lancer.")

;;â
```

8.4 Un compilateur VCMC 2

Voici maintenant un compilateur VLISP pour la machine VCMC2. Voici le texte de ce compilateur :

```
;                C O M P I L   .   V L I                  ;
;                                                          ;
;             Compilateur  VCMC2 en  VLISP 8 . 2          ;
;                ( Taille = 200 doublets )                ;
;----------------------------------------------------------;
;                   Jerome CHAILLOUX                       ;
;                                                          ;
;        Departement d'Informatique                        ;
;        Universite de Paris 8 - Vincennes                 ;
;        Route de la Tourelle 75571 Paris Cedex 12         ;
;        Tel : 374 12 50 poste 299                         ;
;                                                          ;
;        I.R.C.A.M.                                        ;
;        31 Rue St Merri 75004 Paris                       ;
;        Tel : 277 12 33 poste 48-48                       ;
;----------------------------------------------------------;

(STATUS TOPLEVEL 2)  ; Silence durant la lecture


;-----   Les macros caracteres de lecture de liste

(DMC "[" (l x) (UNTIL (EQ (SETQ x (READ)) "]") (NEWL l x)) (FREVERSE l))

(DMC "]" () "]")


;----- fonction de compilation principale

(DE COMPILE (l ;; lap)
   ; l : est la forme à compiler
   ; lap : contiendra la liste des instructions
   ;
   ; 1) génération du code
   ;
   (COMPEXP l)
   (SETQ lap (REVERSE lap))
   ;
   ; 2) impression du résultat obtenu
```

```
   ;
   (TERPRI)
   (MAPC (LAMBDA (x) (PRINCODE x)) lap)
   (TERPRI)
   ;
   ; 3) et voila
   ;
   'OK)


;-----   Phase de macro-génération du code brut


(DE CODE l
   ; ajoute à lap la liste d'instructions l
   (MAPC (LAMBDA (l) (NEWL lap l)) l))


(DE COMPEXP (l)
   ; compile l'expression l
   (COND
        ((NULL l)
           ; la constante NIL
           (CODE '(MOVE NIL A1)))
        ((NUMBP l)
           ; les constantes numériques
           (CODE ['MOVE [QUOTE l] 'A1]))
        ((ATOM l)
           ; les symboles atomiques
           (CODE ['CVAL [QUOTE l] 'A1]))
        ((EQ (CAR l) QUOTE)
           ; la fonction QUOTE
           (CODE ['MOVE [QUOTE (CADR l)] 'A1]))
        ((EQ (CAR l) 'DE)
           ; la fonction de définition DE
           (CODE (CADR l)
              ['MOVE [QUOTE [(CADR l) . (CADDR l)]] 'A4 '[CALL (CBIND)]])
           (FTYPE (CADR l) (SELECTQ (LENGTH (CADDR l))
                (0 '0SUBR) (1 '1SUBR) (2 '2SUBR) (T 'NSUBR)))
           (COMPROGN (CDDDR l))
           (CODE '(NOP () () [RETURN])))
        ((EQ (CAR l) 'IF)
           ; la fonction conditionnelle IF
           (LET ((et1 (GENSYM)) (et2 (GENSYM)))
                (COMPEXP (CADR l))
                (CODE ['TNIL 'A1 () ['LIST 'JUMP [et1]]])
                (COMPEXP (CADDR l))
                (CODE ['NOP () () ['LIST 'JUMP [et2]] et1)
                (COMPROGN (CDDDR l))
                (CODE et2)))
        ((EQ (CAR l) 'PROGN)
            ; le séquenceur PROGN
            (COMPROGN (CDR l)))
        (T (SELECTQ (FTYPE (CAR l))
             ; les fonctions standards de type xSUBR
             (0SUBR
                (CODE ['NOP () () ['LIST 'CALL [(CAR l)]]]))
             (1SUBR
                (COMPEXP (CADR l))
                (CODE ['NOP () () ['LIST 'CALL [(CAR l)]]]))
             (2SUBR
                (COMPEXP (CADR l))
                (CODE '(MOVE A1 TST))
```

```
                (COMPEXP (CADDR l))
                (CODE '(MOVE A1 A2)
                   ['MOVE 'TST 'A1 ['LIST 'CALL [(CAR l)]]]))
             (3SUBR
                (COMPEXP (CADR l))
                (CODE '(MOVE A1 TST))
                (COMPEXP (CADDR l))
                (CODE '(MOVE A1 TST))
                (COMPEXP (CADDDR l))
                (CODE '(MOVE A1 A3) '(MOVE TST A2)
                   ['MOVE 'TST 'A1 ['LIST 'CALL [(CAR l)]]]))
             (NSUBR
                (LET ((narg 0) (larg (CDR l)))
                   (COMPEXP (CAR larg))
                   (IF (NULL (CDR larg))
                      (PROGN
                         (IF (ZEROP narg)
                           (CODE ['XCONS () 'A1 ['LIST 'CALL [(CAR l)]]])
                           (CODE '(XCONS NIL A1))
                           (REPEAT (SUB1 narg) (CODE '(CONS TST A1)))
                           (CODE ['CONS 'TST 'A1 ['LIST 'CALL [(CAR l)]]])))
                      (CODE '(MOVE A1 TST))
                      (SELF (ADD1 narg) (CDR larg)))))
             (T (CODE ['MOVE [QUOTE l] 'A1 '[CALL (EVAL)]]))))))


(DE COMPROGN (l)
   ; compile le progn l
   (IF (NULL l)
      (CODE '(MOVE NIL A1))
      (LET (l l)
         (IF (NULL l)
            ()
            (COMPEXP (CAR l))
            (SELF (CDR l))))))


;---- Fonction d'impression d'une instruction

(DE PRINTCODE (s)
   (IF (ATOM l) (EXIT (PRINT s)))
   (TTAB 10)
   (PRIN (NEXTL s))
   (IF (NULL s) (EXIT (TERPRI)) (TTAB 16))
   (IF (NEQ (CAAR s) 'LIST) (PRIN (NEXTL s)))
   (IF (NULL s) (EXIT (TERPRI)) (PRIN ","))
   (IF (NEQ (CAAR s) 'LIST) (PRIN (NEXTL s)))
   (IF (NULL s) (EXIT (TERPRI)) (PRIN ","))
   (PRIN "[" (CADR s))
   (IF (CDDR s) (PRIN (CADDR s)))
   (PRINT "]"))


;----- Exemples de compilation


(COMPIL '
  (DE LAST (l)
     (IF (NULL (CDR l))
        (CAR l)
        (LAST (CDR l)))))
```

TABLE D'INDEX du MANUEL VLISP 8

